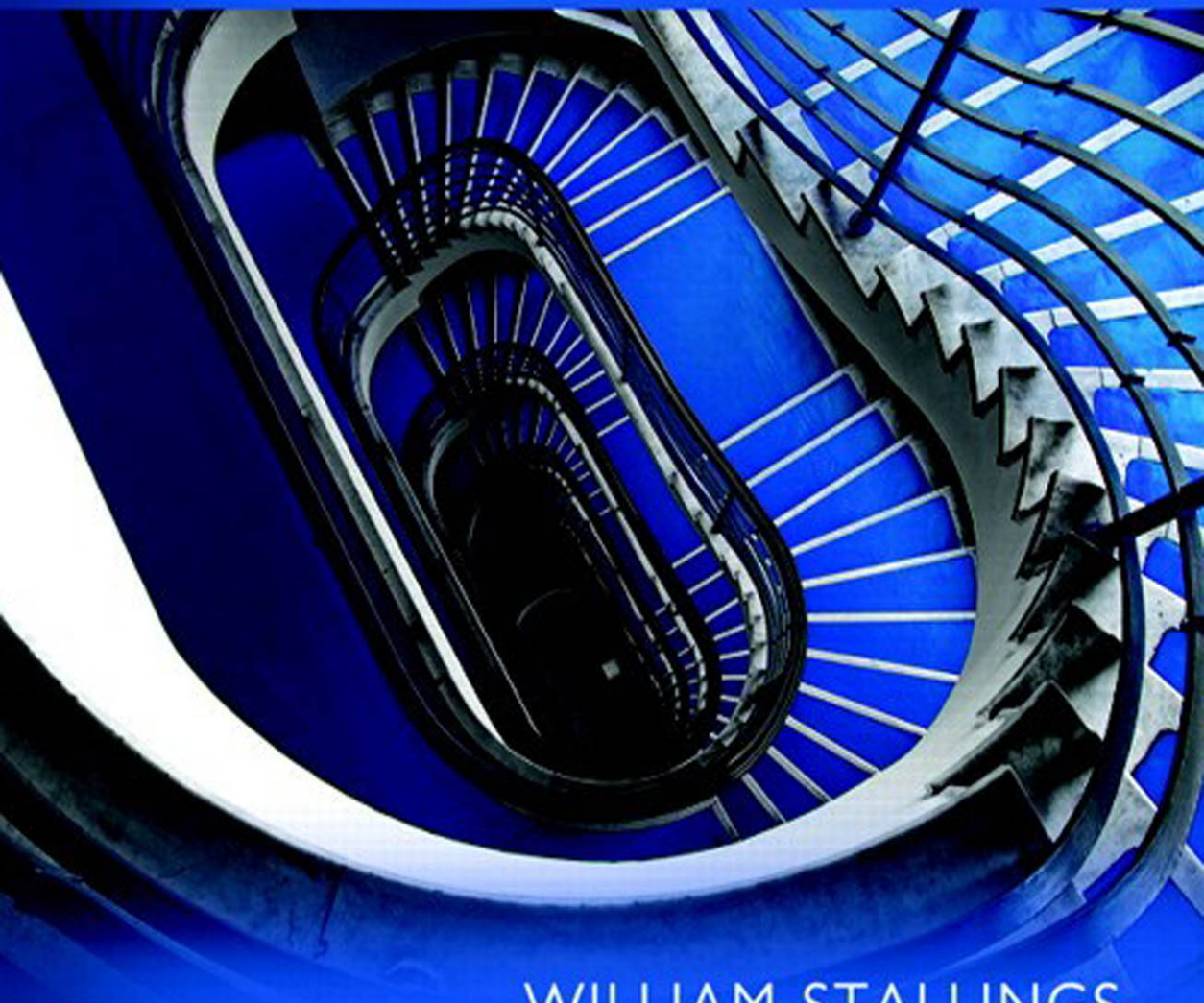
COMPUTER ORGANIZATION AND ARCHITECTURE

Designing for Performance

Tenth Edition



WILLIAM STALLINGS

VISIT…

Computer Organization and Architecture

Designing for Performance

Tenth Edition

This page intentionally left blank

Computer Organization and Architecture

Designing for Performance

Tenth Edition

William Stallings

With contribution by
Peter Zeno
University of Bridgeport

With Foreword by
Chris Jesshope
Professor (emeritus) University of Amsterdam

PEARSON

Boston • Columbus • Hoboken • Indianapolis • New York • San Francisco
Amsterdam • Cape Town • Dubai • London • Madrid • Milan • Munich • Paris • Montreal
Toronto • Delhi • Mexico City • São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Vice President and Editorial Director, ECS: *Marcia J. Horton*
Executive Editor: *Tracy Johnson (Dunkelberger)*
Editorial Assistant: *Kelsey Loanes*
Program Manager: *Carole Snyder*
Director of Product Management: *Erin Gregg*
Team Lead Product Management: *Scott Disanno*
Project Manager: *Robert Engelhardt*
Media Team Lead: *Steve Wright*
R&P Manager: *Rachel Youdelman*
R&P Senior Project Manager: *Timothy Nicholls*
Procurement Manager: *Mary Fischer*
Senior Specialist, Program Planning and Support: *Maura Zaldivar-Garcia*
Inventory Manager: *Bruce Boundy*
VP of Marketing: *Christy Lesko*
Director of Field Marketing: *Demetrius Hall*
Product Marketing Manager: *Bram van Kempen*
Marketing Assistant: *Jon Bryant*
Cover Designer: *Marta Samsel*
Cover Art: © *anderm / Fotolia*
Full-Service Project Management: *Mahalatchoumy Saravanan, Jouve India*
Printer/Binder: *Edwards Brothers Malloy*
Cover Printer: *Lehigh-Phoenix Color/Hagerstown*
Typeface: *Times Ten LT Std 10/12*

Copyright © 2016, 2013, 2010 Pearson Education, Inc., Hoboken, NJ 07030. All rights reserved. Manufactured in the United States of America. This publication is protected by Copyright and permissions should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission(s) to use materials from this work, please submit a written request to Pearson Higher Education, Permissions Department, 221 River Street, Hoboken, NJ 07030.

Many of the designations by manufacturers and seller to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in initial caps or all caps. Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appears on page 833.

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages with, or arising out of, the furnishing, performance, or use of these programs.

Pearson Education Ltd., *London*
Pearson Education Australia Ply. Ltd., Sydney
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd., *Hong Kong*
Pearson Education Canada, Inc., Toronto
Pearson Education de Mexico, S.A. de C.V.
Pearson Education–Japan, Tokyo
Pearson Education Malaysia, Pte. Ltd.
Pearson Education, Inc., *Hoboken, New Jersey*

Library of Congress Cataloging-in-Publication Data

Stallings, William.
Computer organization and architecture : designing for performance / William Stallings. — Tenth edition.
pages cm
Includes bibliographical references and index.
ISBN 978-0-13-410161-3 — ISBN 0-13-410161-8 1. Computer organization. 2. Computer architecture. I. Title.
QA76.9.C643S73 2016
004.2'2—dc23

2014044367

10 9 8 7 6 5 4 3 2 1

PEARSON

www.pearsonhighered.com

ISBN-10: 0-13-410161-8
ISBN-13: 978-0-13-410161-3

To Tricia
my loving wife, the kindest
and gentlest person

This page intentionally left blank

CONTENTS

Foreword xiii

Preface xv

About the Author xxiii

PART ONE INTRODUCTION 1

Chapter 1 Basic Concepts and Computer Evolution 1

1.1 Organization and Architecture 2
1.2 Structure and Function 3
1.3 A Brief History of Computers 11
1.4 The Evolution of the Intel x86 Architecture 27
1.5 Embedded Systems 29
1.6 Arm Architecture 33
1.7 Cloud Computing 39
1.8 Key Terms, Review Questions, and Problems 42

Chapter 2 Performance Issues 45

2.1 Designing for Performance 46
2.2 Multicore, Mics, and GPGPUs 52
2.3 Two Laws that Provide Insight: Ahmdahl's Law and Little's Law 53
2.4 Basic Measures of Computer Performance 56
2.5 Calculating the Mean 59
2.6 Benchmarks and Spec 67
2.7 Key Terms, Review Questions, and Problems 74

PART TWO THE COMPUTER SYSTEM 80

Chapter 3 A Top-Level View of Computer Function and Interconnection 80

3.1 Computer Components 81
3.2 Computer Function 83
3.3 Interconnection Structures 99
3.4 Bus Interconnection 100
3.5 Point-to-Point Interconnect 102
3.6 PCI Express 107
3.7 Key Terms, Review Questions, and Problems 116

Chapter 4 Cache Memory 120

4.1 Computer Memory System Overview 121
4.2 Cache Memory Principles 128
4.3 Elements of Cache Design 131
4.4 Pentium 4 Cache Organization 149
4.5 Key Terms, Review Questions, and Problems 152
Appendix 4A Performance Characteristics of Two-Level Memories 157

Chapter 5 Internal Memory 165

5.1 Semiconductor Main Memory 166
5.2 Error Correction 174
5.3 DDR DRAM 180
5.4 Flash Memory 185
5.5 Newer Nonvolatile Solid-State Memory Technologies 187
5.6 Key Terms, Review Questions, and Problems 190

Chapter 6 External Memory 194

6.1 Magnetic Disk 195
6.2 RAID 204
6.3 Solid State Drives 212
6.4 Optical Memory 217
6.5 Magnetic Tape 222
6.6 Key Terms, Review Questions, and Problems 224

Chapter 7 Input/Output 228

7.1 External Devices 230
7.2 I/O Modules 232
7.3 Programmed I/O 235
7.4 Interrupt-Driven I/O 239
7.5 Direct Memory Access 248
7.6 Direct Cache Access 254
7.7 I/O Channels and Processors 261
7.8 External Interconnection Standards 263
7.9 IBM zEnterprise EC12 I/O Structure 266
7.10 Key Terms, Review Questions, and Problems 270

Chapter 8 Operating System Support 275

8.1 Operating System Overview 276
8.2 Scheduling 287
8.3 Memory Management 293
8.4 Intel x86 Memory Management 304
8.5 Arm Memory Management 309
8.6 Key Terms, Review Questions, and Problems 314

PART THREE ARITHMETIC AND LOGIC 318

Chapter 9 Number Systems 318

9.1 The Decimal System 319
9.2 Positional Number Systems 320
9.3 The Binary System 321
9.4 Converting Between Binary and Decimal 321
9.5 Hexadecimal Notation 324
9.6 Key Terms and Problems 326

Chapter 10 Computer Arithmetic 328

10.1 The Arithmetic and Logic Unit 329
10.2 Integer Representation 330
10.3 Integer Arithmetic 335

10.4 Floating-Point Representation 350
10.5 Floating-Point Arithmetic 358
10.6 Key Terms, Review Questions, and Problems 367

Chapter 11 Digital Logic 372

11.1 Boolean Algebra 373
11.2 Gates 376
11.3 Combinational Circuits 378
11.4 Sequential Circuits 396
11.5 Programmable Logic Devices 405
11.6 Key Terms and Problems 409

PART FOUR THE CENTRAL PROCESSING UNIT 412

Chapter 12 Instruction Sets: Characteristics and Functions 412

12.1 Machine Instruction Characteristics 413
12.2 Types of Operands 420
12.3 Intel x86 and ARM Data Types 422
12.4 Types of Operations 425
12.5 Intel x86 and ARM Operation Types 438
12.6 Key Terms, Review Questions, and Problems 446
Appendix 12A Little-, Big-, and Bi-Endian 452

Chapter 13 Instruction Sets: Addressing Modes and Formats 456

13.1 Addressing Modes 457
13.2 x86 and ARM Addressing Modes 463
13.3 Instruction Formats 469
13.4 x86 and ARM Instruction Formats 477
13.5 Assembly Language 482
13.6 Key Terms, Review Questions, and Problems 484

Chapter 14 Processor Structure and Function 488

14.1 Processor Organization 489
14.2 Register Organization 491
14.3 Instruction Cycle 496
14.4 Instruction Pipelining 500
14.5 The x86 Processor Family 517
14.6 The ARM Processor 524
14.7 Key Terms, Review Questions, and Problems 530

Chapter 15 Reduced Instruction Set Computers 535

15.1 Instruction Execution Characteristics 537
15.2 The Use of a Large Register File 542
15.3 Compiler-Based Register Optimization 547
15.4 Reduced Instruction Set Architecture 549
15.5 RISC Pipelining 555
15.6 MIPS R4000 559
15.7 SPARC 565
15.8 RISC versus CISC Controversy 570
15.9 Key Terms, Review Questions, and Problems 571

Chapter 16 Instruction-Level Parallelism and Superscalar Processors 575

16.1 Overview 576
16.2 Design Issues 581
16.3 Intel Core Microarchitecture 591
16.4 ARM Cortex-A8 596
16.5 ARM Cortex-M3 604
16.6 Key Terms, Review Questions, and Problems 608

PART FIVE PARALLEL ORGANIZATION 613

Chapter 17 Parallel Processing 613

17.1 Multiple Processor Organizations 615
17.2 Symmetric Multiprocessors 617
17.3 Cache Coherence and the MESI Protocol 621
17.4 Multithreading and Chip Multiprocessors 628
17.5 Clusters 633
17.6 Nonuniform Memory Access 640
17.7 Cloud Computing 643
17.8 Key Terms, Review Questions, and Problems 650

Chapter 18 Multicore Computers 656

18.1 Hardware Performance Issues 657
18.2 Software Performance Issues 660
18.3 Multicore Organization 665
18.4 Heterogeneous Multicore Organization 667
18.5 Intel Core i7-990X 676
18.6 ARM Cortex-A15 MPCore 677
18.7 IBM zEnterprise EC12 Mainframe 682
18.8 Key Terms, Review Questions, and Problems 685

Chapter 19 General-Purpose Graphic Processing Units 688

19.1 Cuda Basics 689
19.2 GPU versus CPU 691
19.3 GPU Architecture Overview 692
19.4 Intel's Gen8 GPU 701
19.5 When to Use a GPU as a Coprocessor 704
19.6 Key Terms and Review Questions 706

PART SIX THE CONTROL UNIT 707

Chapter 20 Control Unit Operation 707

20.1 Micro-Operations 708
20.2 Control of the Processor 714
20.3 Hardwired Implementation 724
20.4 Key Terms, Review Questions, and Problems 727

Chapter 21 Microprogrammed Control 729

21.1 Basic Concepts 730
21.2 Microinstruction Sequencing 739

21.3 Microinstruction Execution 745
21.4 TI 8800 755
21.5 Key Terms, Review Questions, and Problems 766

Appendix A Projects for Teaching Computer Organization and Architecture 768

A.1 Interactive Simulations 769
A.2 Research Projects 771
A.3 Simulation Projects 771
A.4 Assembly Language Projects 772
A.5 Reading/Report Assignments 773
A.6 Writing Assignments 773
A.7 Test Bank 773

Appendix B Assembly Language and Related Topics 774

B.1 Assembly Language 775
B.2 Assemblers 783
B.3 Loading and Linking 787
B.4 Key Terms, Review Questions, and Problems 795

References 800

Index 809

Credits 833

ONLINE APPENDICES[1]

Appendix C System Buses
Appendix D Protocols and Protocol Architectures
Appendix E Scrambling
Appendix F Victim Cache Strategies
Appendix G Interleaved Memory
Appendix H International Reference Alphabet
Appendix I Stacks
Appendix J Thunderbolt and Infiniband
Appendix K Virtual Memory Page Replacement Algorithms
Appendix L Hash Tables
Appendix M Recursive Procedures
Appendix N Additional Instruction Pipeline Topics
Appendix O Timing Diagrams
Glossary

[1]Online chapters, appendices, and other documents are Premium Content, available via the access card at the front of this book.

This page intentionally left blank

FOREWORD

by Chris Jesshope
Professor (emeritus) University of Amsterdam
Author of Parallel Computers (with R W Hockney), 1981 & 1988

Having been active in computer organization and architecture for many years, it is a pleasure to write this foreword for the new edition of William Stallings' comprehensive book on this subject. In doing this, I found myself reflecting on the trends and changes in this subject over the time that I have been involved in it. I myself became interested in computer architecture at a time of significant innovation and disruption. That disruption was brought about not only through advances in technology but perhaps more significantly through access to that technology. VLSI was here and VLSI design was available to students in the classroom. These were exciting times. The ability to integrate a mainframe style computer on a single silicon chip was a milestone, but that this was accomplished by an academic research team made the achievement quite unique. This period was characterized by innovation and diversity in computer architecture with one of the main trends being in the area of parallelism. In the 1970s, I had hands-on experience of the Illiac IV, which was an early example of explicit parallelism in computer architecture and which incidentally pioneered all semiconductor memory. This interaction, and it certainly was that, kick-started my own interest in computer architecture and organization, with particular emphasis on explicit parallelism in computer architecture.

Throughout the 1980s and early 1990s research flourished in this field and there was a great deal of innovation, much of which came to market through university start-ups. Ironically however, it was the same technology that reversed this trend. Diversity was gradually replaced with a near monoculture in computer systems with advances in just a few instruction set architectures. Moore's law, a self-fulfilling prediction that became an industry guideline, meant that basic device speeds and integration densities both grew exponentially, with the latter doubling every 18 months of so. The speed increase was the proverbial free lunch for computer architects and the integration levels allowed more complexity and innovation at the micro-architecture level. The free lunch of course did have a cost, that being the exponential growth of capital investment required to fulfill Moore's law, which once again limited the access to state-of-the-art technologies. Moreover, most users found it easier to wait for the next generation of mainstream processor than to invest in the innovations in parallel computers, with their pitfalls and difficulties. The exceptions to this were the few large institutions requiring ultimate performance; two topical examples being large-scale scientific simulation such as climate modeling and also in our security services for code breaking. For

everyone else, the name of the game was compatibility and two instruction set architectures that benefited from this were x86 and ARM, the latter in embedded systems and the former in just about everything else. Parallelism was still there in the implementation of these ISAs, it was just that it was implicit, harnessed by the architecture not in the instruction stream that drives it.

Throughout the late 1990s and early 2000s, this approach to implicitly exploiting concurrency in single-core computer systems flourished. However, in spite of the exponential growth of logic density, it was the cost of the techniques exploited which brought this era to a close. In superscalar processors, the logic costs do not grow linearly with issue width (parallelism), while some components grow as the square or even the cube of the issue width. Although the exponential growth in logic could sustain this continued development, there were two major pitfalls: it was increasingly difficult to expose concurrency implicitly from imperative programs and hence efficiencies in the use of instruction issue slots decreased. Perhaps more importantly, technology was experiencing a new barrier to performance gains, namely that of power dissipation, and several superscalar developments were halted because the silicon in them would have been too hot. These constraints have mandated the exploitation of explicit parallelism, despite the compatibility challenges. So it seems that again innovation and diversity are opening up this area to new research.

Perhaps not since the 1980s has it been so interesting to study in this field. That diversity is an economic reality can be seen by the decrease in issue width (implicit parallelism) and increase in the number of cores (explicit parallelism) in mainstream processors. However, the question is how to exploit this, both at the application and the system level. There are significant challenges here still to be solved. Superscalar processors rely on the processor to extract parallelism from a single instruction stream. What if we shifted the emphasis and provided an instruction stream with maximum parallelism, how can we exploit this in different configurations and/or generations of processors that require different levels of explicit parallelism? Is it possible therefore to have a micro-architecture that sequentializes and schedules this maximum concurrency captured in the ISA to match the current configuration of cores so that we gain the same compatibility in a world of explicit parallelism? Does this require operating systems in silicon for efficiency?

These are just some of the questions facing us today. To answer these questions and more requires a sound foundation in computer organization and architecture, and this book by William Stallings provides a very timely and comprehensive foundation. It gives a complete introduction to the basics required, tackling what can be quite complex topics with apparent simplicity. Moreover, it deals with the more recent developments in this field, where innovation has in the past, and is, currently taking place. Examples are in superscalar issue and in explicitly parallel multicores. What is more, this latest edition includes two very recent topics in the design and use of GPUs for general-purpose use and the latest trends in cloud computing, both of which have become mainstream only recently. The book makes good use of examples throughout to highlight the theoretical issues covered, and most of these examples are drawn from developments in the two most widely used ISAs, namely the x86 and ARM. To reiterate, this book is complete and is a pleasure to read and hopefully will kick-start more young researchers down the same path that I have enjoyed over the last 40 years!

PREFACE

WHAT'S NEW IN THE TENTH EDITION

Since the ninth edition of this book was published, the field has seen continued innovations and improvements. In this new edition, I try to capture these changes while maintaining a broad and comprehensive coverage of the entire field. To begin this process of revision, the ninth edition of this book was extensively reviewed by a number of professors who teach the subject and by professionals working in the field. The result is that, in many places, the narrative has been clarified and tightened, and illustrations have been improved.

Beyond these refinements to improve pedagogy and user-friendliness, there have been substantive changes throughout the book. Roughly the same chapter organization has been retained, but much of the material has been revised and new material has been added. The most noteworthy changes are as follows:

- **GPGPU [General-Purpose Computing on Graphics Processing Units (GPUs)]:** One of the most important new developments in recent years has been the broad adoption of GPGPUs to work in coordination with traditional CPUs to handle a wide range of applications involving large arrays of data. A new chapter is devoted to the topic of GPGPUs.
- **Heterogeneous multicore processors:** The latest development in multicore architecture is the heterogeneous multicore processor. A new section in the chapter on multicore processors surveys the various types of heterogeneous multicore processors.
- **Embedded systems:** The overview of embedded systems in Chapter 1 has been substantially revised and expanded to reflect the current state of embedded technology.
- **Microcontrollers:** In terms of numbers, almost all computers now in use are embedded microcontrollers. The treatment of embedded systems in Chapter 1 now includes coverage of microcontrollers. The ARM Cortex-M3 microcontroller is used as an example system throughout the text.
- **Cloud computing:** New to this edition is a discussion of cloud computing, with an overview in Chapter 1 and more detailed treatment in Chapter 17.
- **System performance:** The coverage of system performance issues has been revised, expanded, and reorganized for a clearer and more thorough treatment. Chapter 2 is devoted to this topic, and the issue of system performance arises throughout the book.

- **Flash memory:** The coverage of flash memory has been updated and expanded, and now includes a discussion of the technology and organization of flash memory for internal memory (Chapter 5) and external memory (Chapter 6).
- **Nonvolatile RAM:** New to this edition is treatment of three important new nonvolatile solid-state RAM technologies that occupy different positions in the memory hierarchy: STT-RAM, PCRAM, and ReRAM.
- **Direct cache access (DCA):** To meet the protocol processing demands for very high speed network connections, Intel and other manufacturers have developed DCA technologies that provide much greater throughput than traditional direct memory access (DMA) approaches. New to this edition, Chapter 7 explores DCA in some detail.
- **Intel Core Microarchitecture:** As in the previous edition, the Intel x86 family is used as a major example system throughout. The treatment has been updated to reflect newer Intel systems, especially the Intel Core Microarchitecture, which is used on both PC and server products.
- **Homework problems:** The number of supplemental homework problems, with solutions, available for student practice has been expanded.

SUPPORT OF ACM/IEEE COMPUTER SCIENCE CURRICULA 2013

The book is intended for both an academic and a professional audience. As a textbook, it is intended as a one- or two-semester undergraduate course for computer science, computer engineering, and electrical engineering majors. This edition is designed to support the recommendations of the ACM/IEEE Computer Science Curricula 2013 (CS2013). CS2013 divides all course work into three categories: Core-Tier 1 (all topics should be included in the curriculum); Core-Tier-2 (all or almost all topics should be included); and Elective (desirable to provide breadth and depth). In the Architecture and Organization (AR) area, CS2013 includes five Tier-2 topics and three Elective topics, each of which has a number of subtopics. This text covers all eight topics listed by CS2013. Table P.1 shows the support for the AR Knowledge Area provided in this textbook.

Table P.1 Coverage of CS2013 Architecture and Organization (AR) Knowledge Area

IAS Knowledge Units	Topics	Textbook Coverage
Digital Logic and Digital Systems (Tier 2)	• Overview and history of computer architecture • Combinational vs. sequential logic/Field programmable gate arrays as a fundamental combinational sequential logic building block • Multiple representations/layers of interpretation (hardware is just another layer) • Physical constraints (gate delays, fan-in, fan-out, energy/power)	—Chapter 1 —Chapter 11
Machine Level Representation of Data (Tier 2)	• Bits, bytes, and words • Numeric data representation and number bases • Fixed- and floating-point systems • Signed and twos-complement representations • Representation of non-numeric data (character codes, graphical data)	—Chapter 9 —Chapter 10

IAS Knowledge Units	Topics	Textbook Coverage
Assembly Level Machine Organization (Tier 2)	• Basic organization of the von Neumann machine • Control unit; instruction fetch, decode, and execution • Instruction sets and types (data manipulation, control, I/O) • Assembly/machine language programming • Instruction formats • Addressing modes • Subroutine call and return mechanisms (cross-reference PL/Language Translation and Execution) • I/O and interrupts • Shared memory multiprocessors/multicore organization • Introduction to SIMD vs. MIMD and the Flynn Taxonomy	—Chapter 1 —Chapter 7 —Chapter 12 —Chapter 13 —Chapter 17 —Chapter 18 —Chapter 20 —Chapter 21 —Appendix A
Memory System Organization and Architecture (Tier 2)	• Storage systems and their technology • Memory hierarchy: temporal and spatial locality • Main memory organization and operations • Latency, cycle time, bandwidth, and interleaving • Cache memories (address mapping, block size, replacement and store policy) • Multiprocessor cache consistency/Using the memory system for inter-core synchronization/atomic memory operations • Virtual memory (page table, TLB) • Fault handling and reliability	—Chapter 4 —Chapter 5 —Chapter 6 —Chapter 8 —Chapter 17
Interfacing and Communication (Tier 2)	• I/O fundamentals: handshaking, buffering, programmed I/O, interrupt-driven I/O • Interrupt structures: vectored and prioritized, interrupt acknowledgment • External storage, physical organization, and drives • Buses: bus protocols, arbitration, direct-memory access (DMA) • RAID architectures	—Chapter 3 —Chapter 6 —Chapter 7
Functional Organization (Elective)	• Implementation of simple datapaths, including instruction pipelining, hazard detection, and resolution • Control unit: hardwired realization vs. microprogrammed realization • Instruction pipelining • Introduction to instruction-level parallelism (ILP)	—Chapter 14 —Chapter 16 —Chapter 20 —Chapter 21
Multiprocessing and Alternative Architectures (Elective)	• Example SIMD and MIMD instruction sets and architectures • Interconnection networks • Shared multiprocessor memory systems and memory consistency • Multiprocessor cache coherence	—Chapter 12 —Chapter 13 —Chapter 17
Performance Enhancements (Elective)	• Superscalar architecture • Branch prediction, Speculative execution, Out-of-order execution • Prefetching • Vector processors and GPUs • Hardware support for multithreading • Scalability	—Chapter 15 —Chapter 16 —Chapter 19

OBJECTIVES

This book is about the structure and function of computers. Its purpose is to present, as clearly and completely as possible, the nature and characteristics of modern-day computer systems.

This task is challenging for several reasons. First, there is a tremendous variety of products that can rightly claim the name of computer, from single-chip microprocessors costing a few dollars to supercomputers costing tens of millions of dollars. Variety is exhibited not only in cost but also in size, performance, and application. Second, the rapid pace of change that has always characterized computer technology continues with no letup. These changes cover all aspects of computer technology, from the underlying integrated circuit technology used to construct computer components to the increasing use of parallel organization concepts in combining those components.

In spite of the variety and pace of change in the computer field, certain fundamental concepts apply consistently throughout. The application of these concepts depends on the current state of the technology and the price/performance objectives of the designer. The intent of this book is to provide a thorough discussion of the fundamentals of computer organization and architecture and to relate these to contemporary design issues.

The subtitle suggests the theme and the approach taken in this book. It has always been important to design computer systems to achieve high performance, but never has this requirement been stronger or more difficult to satisfy than today. All of the basic performance characteristics of computer systems, including processor speed, memory speed, memory capacity, and interconnection data rates, are increasing rapidly. Moreover, they are increasing at different rates. This makes it difficult to design a balanced system that maximizes the performance and utilization of all elements. Thus, computer design increasingly becomes a game of changing the structure or function in one area to compensate for a performance mismatch in another area. We will see this game played out in numerous design decisions throughout the book.

A computer system, like any system, consists of an interrelated set of components. The system is best characterized in terms of structure—the way in which components are interconnected, and function—the operation of the individual components. Furthermore, a computer's organization is hierarchical. Each major component can be further described by decomposing it into its major subcomponents and describing their structure and function. For clarity and ease of understanding, this hierarchical organization is described in this book from the top down:

- **Computer system:** Major components are processor, memory, I/O.
- **Processor:** Major components are control unit, registers, ALU, and instruction execution unit.
- **Control unit:** Provides control signals for the operation and coordination of all processor components. Traditionally, a microprogramming implementation has been used, in which major components are control memory, microinstruction sequencing logic, and registers. More recently, microprogramming has been less prominent but remains an important implementation technique.

The objective is to present the material in a fashion that keeps new material in a clear context. This should minimize the chance that the reader will get lost and should provide better motivation than a bottom-up approach.

Throughout the discussion, aspects of the system are viewed from the points of view of both architecture (those attributes of a system visible to a machine language programmer) and organization (the operational units and their interconnections that realize the architecture).

EXAMPLE SYSTEMS

This text is intended to acquaint the reader with the design principles and implementation issues of contemporary operating systems. Accordingly, a purely conceptual or theoretical treatment would be inadequate. To illustrate the concepts and to tie them to real-world design choices that must be made, two processor families have been chosen as running examples:

- **Intel x86 architecture:** The x86 architecture is the most widely used for nonembedded computer systems. The x86 is essentially a complex instruction set computer (CISC) with some RISC features. Recent members of the x86 family make use of superscalar and multicore design principles. The evolution of features in the x86 architecture provides a unique case-study of the evolution of most of the design principles in computer architecture.
- **ARM:** The ARM architecture is arguably the most widely used embedded processor, used in cell phones, iPods, remote sensor equipment, and many other devices. The ARM is essentially a reduced instruction set computer (RISC). Recent members of the ARM family make use of superscalar and multicore design principles.

Many, but by no means all, of the examples in this book are drawn from these two computer families. Numerous other systems, both contemporary and historical, provide examples of important computer architecture design features.

PLAN OF THE TEXT

The book is organized into six parts:

- Overview
- The computer system
- Arithmetic and logic
- The central processing unit
- Parallel organization, including multicore
- The control unit

The book includes a number of pedagogic features, including the use of interactive simulations and numerous figures and tables to clarify the discussion. Each chapter includes a list of key words, review questions, homework problems, and suggestions for further reading. The book also includes an extensive glossary, a list of frequently used acronyms, and a bibliography.

INSTRUCTOR SUPPORT MATERIALS

Support materials for instructors are available at the **Instructor Resource Center (IRC)** for this textbook, which can be reached through the publisher's Web site www.pearsonhighered.com/stallings or by clicking on the link labeled "Pearson Resources for Instructors" at this

book's Companion Web site at WilliamStallings.com/ComputerOrganization. To gain access to the IRC, please contact your local Pearson sales representative via pearsonhighered.com/educator/replocator/requestSalesRep.page or call Pearson Faculty Services at 1-800-526-0485. The IRC provides the following materials:

- **Projects manual:** Project resources including documents and portable software, plus suggested project assignments for all of the project categories listed subsequently in this Preface.
- **Solutions manual:** Solutions to end-of-chapter Review Questions and Problems.
- **PowerPoint slides:** A set of slides covering all chapters, suitable for use in lecturing.
- **PDF files:** Copies of all figures and tables from the book.
- **Test bank:** A chapter-by-chapter set of questions.
- **Sample syllabuses:** The text contains more material than can be conveniently covered in one semester. Accordingly, instructors are provided with several sample syllabuses that guide the use of the text within limited time. These samples are based on real-world experience by professors with the first edition.

The **Companion Web site**, at WilliamStallings.com/ComputerOrganization (click on Instructor Resources link) includes the following:

- Links to Web sites for other courses being taught using this book.
- Sign-up information for an Internet mailing list for instructors using this book to exchange information, suggestions, and questions with each other and with the author.

STUDENT RESOURCES

For this new edition, a tremendous amount of original supporting material for students has been made available online, at two Web locations. The **Companion Web Site**, at WilliamStallings.com/ComputerOrganization (click on Student Resources link), includes a list of relevant links organized by chapter and an errata sheet for the book.

Purchasing this textbook new grants the reader six months of access to the **Premium Content Site**, which includes the following materials:

- **Online chapters:** To limit the size and cost of the book, two chapters of the book are provided in PDF format. The chapters are listed in this book's table of contents.
- **Online appendices:** There are numerous interesting topics that support material found in the text but whose inclusion is not warranted in the printed text. A total of 13 appendices cover these topics for the interested student. The appendices are listed in this book's table of contents.
- **Homework problems and solutions:** To aid the student in understanding the material, a separate set of homework problems with solutions are available. Students can enhance their understanding of the material by working out the solutions to these problems and then checking their answers.

To access the Premium Content site, click on the *Premium Content* link at the Companion Web site or at pearsonhighered.com/stallings and enter the student access code found on the card in the front of the book.

Finally, I maintain the Computer Science Student Resource Site at **WilliamStallings.com/StudentSupport.html**.

PROJECTS AND OTHER STUDENT EXERCISES

For many instructors, an important component of a computer organization and architecture course is a project or set of projects by which the student gets hands-on experience to reinforce concepts from the text. This book provides an unparalleled degree of support for including a projects component in the course. The instructor's support materials available through Prentice Hall not only includes guidance on how to assign and structure the projects but also includes a set of user's manuals for various project types plus specific assignments, all written especially for this book. Instructors can assign work in the following areas:

- **Interactive simulation assignments:** Described subsequently.
- **Research projects:** A series of research assignments that instruct the student to research a particular topic on the Internet and write a report.
- **Simulation projects:** The IRC provides support for the use of the two simulation packages: SimpleScalar can be used to explore computer organization and architecture design issues. SMPCache provides a powerful educational tool for examining cache design issues for symmetric multiprocessors.
- **Assembly language projects:** A simplified assembly language, CodeBlue, is used and assignments based on the popular Core Wars concept are provided.
- **Reading/report assignments:** A list of papers in the literature, one or more for each chapter, that can be assigned for the student to read and then write a short report.
- **Writing assignments:** A list of writing assignments to facilitate learning the material.
- **Test bank:** Includes T/F, multiple choice, and fill-in-the-blank questions and answers.

This diverse set of projects and other student exercises enables the instructor to use the book as one component in a rich and varied learning experience and to tailor a course plan to meet the specific needs of the instructor and students. See Appendix A in this book for details.

INTERACTIVE SIMULATIONS

An important feature in this edition is the incorporation of interactive simulations. These simulations provide a powerful tool for understanding the complex design features of a modern computer system. A total of 20 interactive simulations are used to illustrate key functions and algorithms in computer organization and architecture design. At the relevant point in the book, an icon indicates that a relevant interactive simulation is available online for student use. Because the animations enable the user to set initial conditions, they can

serve as the basis for student assignments. The instructor's supplement includes a set of assignments, one for each of the animations. Each assignment includes several specific problems that can be assigned to students.

For access to the animations, click on the rotating globe at this book's Web site at http://williamstallings.com/ComputerOrganization.

ACKNOWLEDGMENTS

This new edition has benefited from review by a number of people, who gave generously of their time and expertise. The following professors and instructors reviewed all or a large part of the manuscript: Molisa Derk (Dickinson State University), Yaohang Li (Old Dominion University), Dwayne Ockel (Regis University), Nelson Luiz Passos (Midwestern State University), Mohammad Abdus Salam (Southern University), and Vladimir Zwass (Fairleigh Dickinson University).

Thanks also to the many people who provided detailed technical reviews of one or more chapters: Rekai Gonzalez Alberquilla, Allen Baum, Jalil Boukhobza, Dmitry Bufistov, Humberto Calderón, Jesus Carretero, Ashkan Eghbal, Peter Glaskowsky, Ram Huggahalli, Chris Jesshope, Athanasios Kakarountas, Isil Oz, Mitchell Poplingher, Roger Shepherd, Jigar Savla, Karl Stevens, Siri Uppalapati, Dr. Sriram Vajapeyam, Kugan Vivekanandarajah, Pooria M. Yaghini, and Peter Zeno,

Peter Zeno also contributed Chapter 19 on GPGPUs.

Professor Cindy Norris of Appalachian State University, Professor Bin Mu of the University of New Brunswick, and Professor Kenrick Mock of the University of Alaska kindly supplied homework problems.

Aswin Sreedhar of the University of Massachusetts developed the interactive simulation assignments and also wrote the test bank.

Professor Miguel Angel Vega Rodriguez, Professor Dr. Juan Manuel Sánchez Pérez, and Professor Dr. Juan Antonio Gómez Pulido, all of University of Extremadura, Spain, prepared the SMPCache problems in the instructor's manual and authored the SMPCache User's Guide.

Todd Bezenek of the University of Wisconsin and James Stine of Lehigh University prepared the SimpleScalar problems in the instructor's manual, and Todd also authored the SimpleScalar User's Guide.

Finally, I would like to thank the many people responsible for the publication of the book, all of whom did their usual excellent job. This includes the staff at Pearson, particularly my editor Tracy Johnson, her assistant Kelsey Loanes, program manager Carole Snyder, and production manager Bob Engelhardt. I also thank Mahalatchoumy Saravanan and the production staff at Jouve India for another excellent and rapid job. Thanks also to the marketing and sales staffs at Pearson, without whose efforts this book would not be in front of you.

About the Author

Dr. William Stallings has authored 17 textbooks, and counting revised editions, over 40 books on computer security, computer networking, and computer architecture. In over 30 years in the field, he has been a technical contributor, technical manager, and an executive with several high-technology firms. Currently, he is an independent consultant whose clients have included computer and networking manufacturers and customers, software development firms, and leading-edge government research institutions. He has 13 times received the award for the best computer science textbook of the year from the Text and Academic Authors Association.

He created and maintains the Computer Science Student Resource Site at ComputerScienceStudent.com. This site provides documents and links on a variety of subjects of general interest to computer science students (and professionals). His articles appear regularly at networking.answers.com, where he is the Networking Category Expert Writer. He is a member of the editorial board of *Cryptologia*, a scholarly journal devoted to all aspects of cryptology.

Dr. Stallings holds a PhD from MIT in computer science and a BS from Notre Dame in electrical engineering.

This page intentionally left blank

PART ONE INTRODUCTION

CHAPTER 1

BASIC CONCEPTS AND COMPUTER EVOLUTION

1.1 **Organization and Architecture**

1.2 **Structure and Function**
- Function
- Structure

1.3 **A Brief History of Computers**
- The First Generation: Vacuum Tubes
- The Second Generation: Transistors
- The Third Generation: Integrated Circuits
- Later Generations

1.4 **The Evolution of the Intel x86 Architecture**

1.5 **Embedded Systems**
- The Internet of Things
- Embedded Operating Systems
- Application Processors versus Dedicated Processors
- Microprocessors versus Microcontrollers
- Embedded versus Deeply Embedded Systems

1.6 **ARM Architecture**
- ARM Evolution
- Instruction Set Architecture
- ARM Products

1.7 **Cloud Computing**
- Basic Concepts
- Cloud Services

1.8 **Key Terms, Review Questions, and Problems**

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Explain the general functions and structure of a digital computer.
- Present an overview of the evolution of computer technology from early digital computers to the latest microprocessors.
- Present an overview of the evolution of the x86 architecture.
- Define embedded systems and list some of the requirements and constraints that various embedded systems must meet.

1.1 ORGANIZATION AND ARCHITECTURE

In describing computers, a distinction is often made between *computer architecture* and *computer organization*. Although it is difficult to give precise definitions for these terms, a consensus exists about the general areas covered by each. For example, see [VRAN80], [SIEW82], and [BELL78a]; an interesting alternative view is presented in [REDD76].

Computer architecture refers to those attributes of a system visible to a programmer or, put another way, those attributes that have a direct impact on the logical execution of a program. A term that is often used interchangeably with computer architecture is **instruction set architecture (ISA)**. The ISA defines instruction formats, instruction opcodes, registers, instruction and data memory; the effect of executed instructions on the registers and memory; and an algorithm for controlling instruction execution. **Computer organization** refers to the operational units and their interconnections that realize the architectural specifications. Examples of architectural attributes include the instruction set, the number of bits used to represent various data types (e.g., numbers, characters), I/O mechanisms, and techniques for addressing memory. Organizational attributes include those hardware details transparent to the programmer, such as control signals; interfaces between the computer and peripherals; and the memory technology used.

For example, it is an architectural design issue whether a computer will have a multiply instruction. It is an organizational issue whether that instruction will be implemented by a special multiply unit or by a mechanism that makes repeated use of the add unit of the system. The organizational decision may be based on the anticipated frequency of use of the multiply instruction, the relative speed of the two approaches, and the cost and physical size of a special multiply unit.

Historically, and still today, the distinction between architecture and organization has been an important one. Many computer manufacturers offer a family of computer models, all with the same architecture but with differences in organization. Consequently, the different models in the family have different price and performance characteristics. Furthermore, a particular architecture may span many years and encompass a number of different computer models, its organization changing with changing technology. A prominent example of both these phenomena is the IBM System/370 architecture. This architecture was first introduced in 1970 and

included a number of models. The customer with modest requirements could buy a cheaper, slower model and, if demand increased, later upgrade to a more expensive, faster model without having to abandon software that had already been developed. Over the years, IBM has introduced many new models with improved technology to replace older models, offering the customer greater speed, lower cost, or both. These newer models retained the same architecture so that the customer's software investment was protected. Remarkably, the System/370 architecture, with a few enhancements, has survived to this day as the architecture of IBM's mainframe product line.

In a class of computers called microcomputers, the relationship between architecture and organization is very close. Changes in technology not only influence organization but also result in the introduction of more powerful and more complex architectures. Generally, there is less of a requirement for generation-to-generation compatibility for these smaller machines. Thus, there is more interplay between organizational and architectural design decisions. An intriguing example of this is the reduced instruction set computer (RISC), which we examine in Chapter 15.

This book examines both computer organization and computer architecture. The emphasis is perhaps more on the side of organization. However, because a computer organization must be designed to implement a particular architectural specification, a thorough treatment of organization requires a detailed examination of architecture as well.

1.2 STRUCTURE AND FUNCTION

A computer is a complex system; contemporary computers contain millions of elementary electronic components. How, then, can one clearly describe them? The key is to recognize the hierarchical nature of most complex systems, including the computer [SIMO96]. A hierarchical system is a set of interrelated subsystems, each of the latter, in turn, hierarchical in structure until we reach some lowest level of elementary subsystem.

The hierarchical nature of complex systems is essential to both their design and their description. The designer need only deal with a particular level of the system at a time. At each level, the system consists of a set of components and their interrelationships. The behavior at each level depends only on a simplified, abstracted characterization of the system at the next lower level. At each level, the designer is concerned with structure and function:

- **Structure:** The way in which the components are interrelated.
- **Function:** The operation of each individual component as part of the structure.

In terms of description, we have two choices: starting at the bottom and building up to a complete description, or beginning with a top view and decomposing the system into its subparts. Evidence from a number of fields suggests that the top-down approach is the clearest and most effective [WEIN75].

The approach taken in this book follows from this viewpoint. The computer system will be described from the top down. We begin with the major components of a computer, describing their structure and function, and proceed to successively

lower layers of the hierarchy. The remainder of this section provides a very brief overview of this plan of attack.

Function

Both the structure and functioning of a computer are, in essence, simple. In general terms, there are only four basic functions that a computer can perform:

- **Data processing:** Data may take a wide variety of forms, and the range of processing requirements is broad. However, we shall see that there are only a few fundamental methods or types of data processing.
- **Data storage:** Even if the computer is processing data on the fly (i.e., data come in and get processed, and the results go out immediately), the computer must temporarily store at least those pieces of data that are being worked on at any given moment. Thus, there is at least a short-term data storage function. Equally important, the computer performs a long-term data storage function. Files of data are stored on the computer for subsequent retrieval and update.
- **Data movement:** The computer's operating environment consists of devices that serve as either sources or destinations of data. When data are received from or delivered to a device that is directly connected to the computer, the process is known as *input–output* (*I/O*), and the device is referred to as a *peripheral*. When data are moved over longer distances, to or from a remote device, the process is known as *data communications*.
- **Control:** Within the computer, a control unit manages the computer's resources and orchestrates the performance of its functional parts in response to instructions.

The preceding discussion may seem absurdly generalized. It is certainly possible, even at a top level of computer structure, to differentiate a variety of functions, but to quote [SIEW82]:

> There is remarkably little shaping of computer structure to fit the function to be performed. At the root of this lies the general-purpose nature of computers, in which all the functional specialization occurs at the time of programming and not at the time of design.

Structure

We now look in a general way at the internal structure of a computer. We begin with a traditional computer with a single processor that employs a microprogrammed control unit, then examine a typical multicore structure.

SIMPLE SINGLE-PROCESSOR COMPUTER Figure 1.1 provides a hierarchical view of the internal structure of a traditional single-processor computer. There are four main structural components:

- **Central processing unit (CPU):** Controls the operation of the computer and performs its data processing functions; often simply referred to as **processor**.
- **Main memory:** Stores data.

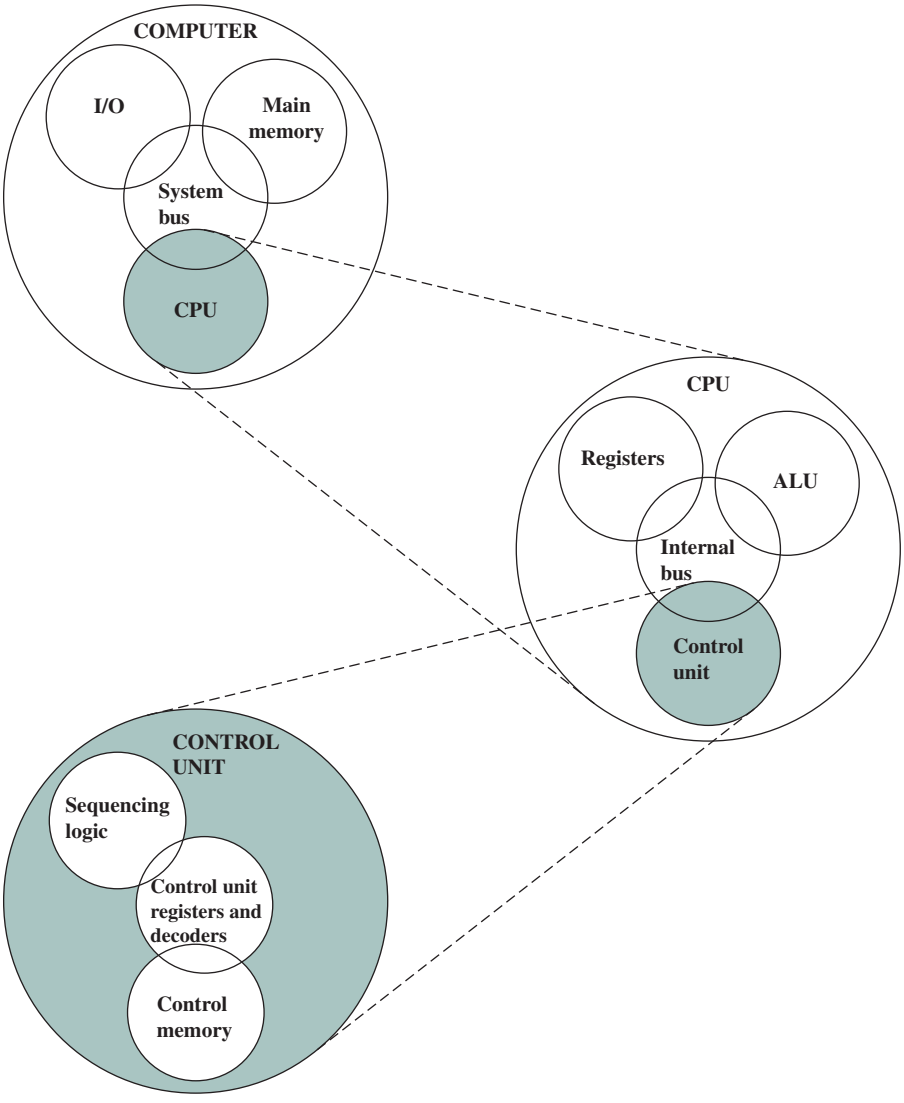


Figure 1.1 The Computer: Top-Level Structure

- **I/O:** Moves data between the computer and its external environment.
- **System interconnection:** Some mechanism that provides for communication among CPU, main memory, and I/O. A common example of system interconnection is by means of a **system bus**, consisting of a number of conducting wires to which all the other components attach.

There may be one or more of each of the aforementioned components. Traditionally, there has been just a single processor. In recent years, there has been increasing use of multiple processors in a single computer. Some design issues relating to multiple processors crop up and are discussed as the text proceeds; Part Five focuses on such computers.

Each of these components will be examined in some detail in Part Two. However, for our purposes, the most interesting and in some ways the most complex component is the CPU. Its major structural components are as follows:

- **Control unit:** Controls the operation of the CPU and hence the computer.
- **Arithmetic and logic unit (ALU):** Performs the computer's data processing functions.
- **Registers:** Provides storage internal to the CPU.
- **CPU interconnection:** Some mechanism that provides for communication among the control unit, ALU, and registers.

Part Three covers these components, where we will see that complexity is added by the use of parallel and pipelined organizational techniques. Finally, there are several approaches to the implementation of the control unit; one common approach is a *microprogrammed* implementation. In essence, a microprogrammed control unit operates by executing microinstructions that define the functionality of the control unit. With this approach, the structure of the control unit can be depicted, as in Figure 1.1. This structure is examined in Part Four.

MULTICORE COMPUTER STRUCTURE As was mentioned, contemporary computers generally have multiple processors. When these processors all reside on a single chip, the term *multicore computer* is used, and each processing unit (consisting of a control unit, ALU, registers, and perhaps cache) is called a *core*. To clarify the terminology, this text will use the following definitions.

- **Central processing unit (CPU):** That portion of a computer that fetches and executes instructions. It consists of an ALU, a control unit, and registers. In a system with a single processing unit, it is often simply referred to as a *processor*.
- **Core:** An individual processing unit on a processor chip. A core may be equivalent in functionality to a CPU on a single-CPU system. Other specialized processing units, such as one optimized for vector and matrix operations, are also referred to as cores.
- **Processor:** A physical piece of silicon containing one or more cores. The processor is the computer component that interprets and executes instructions. If a processor contains multiple cores, it is referred to as a **multicore processor**.

After about a decade of discussion, there is broad industry consensus on this usage.

Another prominent feature of contemporary computers is the use of multiple layers of memory, called *cache memory*, between the processor and main memory. Chapter 4 is devoted to the topic of cache memory. For our purposes in this section, we simply note that a cache memory is smaller and faster than main memory and is used to speed up memory access, by placing in the cache data from main memory, that is likely to be used in the near future. A greater performance improvement may be obtained by using multiple levels of cache, with level 1 (L1) closest to the core and additional levels (L2, L3, and so on) progressively farther from the core. In this scheme, level n is smaller and faster than level $n + 1$.

Figure 1.2 is a simplified view of the principal components of a typical multicore computer. Most computers, including embedded computers in smartphones and tablets, plus personal computers, laptops, and workstations, are housed on a motherboard. Before describing this arrangement, we need to define some terms. A **printed circuit board (PCB)** is a rigid, flat board that holds and interconnects chips and other electronic components. The board is made of layers, typically two to ten, that interconnect components via copper pathways that are etched into the board. The main printed circuit board in a computer is called a system board or **motherboard**, while smaller ones that plug into the slots in the main board are called expansion boards.

The most prominent elements on the motherboard are the chips. A **chip** is a single piece of semiconducting material, typically silicon, upon which electronic circuits and logic gates are fabricated. The resulting product is referred to as an **integrated circuit**.

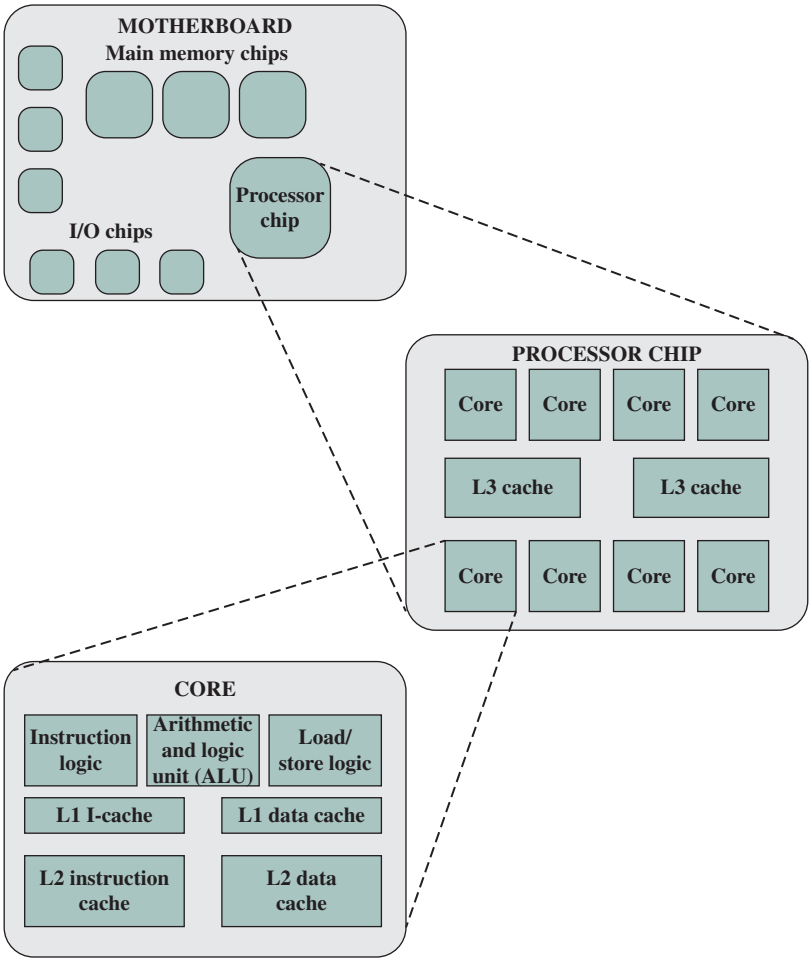


Figure 1.2 Simplified View of Major Elements of a Multicore Computer

The motherboard contains a slot or socket for the processor chip, which typically contains multiple individual cores, in what is known as a *multicore processor*. There are also slots for memory chips, I/O controller chips, and other key computer components. For desktop computers, expansion slots enable the inclusion of more components on expansion boards. Thus, a modern motherboard connects only a few individual chip components, with each chip containing from a few thousand up to hundreds of millions of transistors.

Figure 1.2 shows a processor chip that contains eight cores and an L3 cache. Not shown is the logic required to control operations between the cores and the cache and between the cores and the external circuitry on the motherboard. The figure indicates that the L3 cache occupies two distinct portions of the chip surface. However, typically, all cores have access to the entire L3 cache via the aforementioned control circuits. The processor chip shown in Figure 1.2 does not represent any specific product, but provides a general idea of how such chips are laid out.

Next, we zoom in on the structure of a single core, which occupies a portion of the processor chip. In general terms, the functional elements of a core are:

- **Instruction logic:** This includes the tasks involved in fetching instructions, and decoding each instruction to determine the instruction operation and the memory locations of any operands.
- **Arithmetic and logic unit (ALU):** Performs the operation specified by an instruction.
- **Load/store logic:** Manages the transfer of data to and from main memory via cache.

The core also contains an L1 cache, split between an instruction cache (I-cache) that is used for the transfer of instructions to and from main memory, and an L1 data cache, for the transfer of operands and results. Typically, today's processor chips also include an L2 cache as part of the core. In many cases, this cache is also split between instruction and data caches, although a combined, single L2 cache is also used.

Keep in mind that this representation of the layout of the core is only intended to give a general idea of internal core structure. In a given product, the functional elements may not be laid out as the three distinct elements shown in Figure 1.2, especially if some or all of these functions are implemented as part of a microprogrammed control unit.

EXAMPLES It will be instructive to look at some real-world examples that illustrate the hierarchical structure of computers. Figure 1.3 is a photograph of the motherboard for a computer built around two Intel Quad-Core Xeon processor chips. Many of the elements labeled on the photograph are discussed subsequently in this book. Here, we mention the most important, in addition to the processor sockets:

- PCI-Express slots for a high-end display adapter and for additional peripherals (Section 3.6 describes PCIe).
- Ethernet controller and Ethernet ports for network connections.
- USB sockets for peripheral devices.

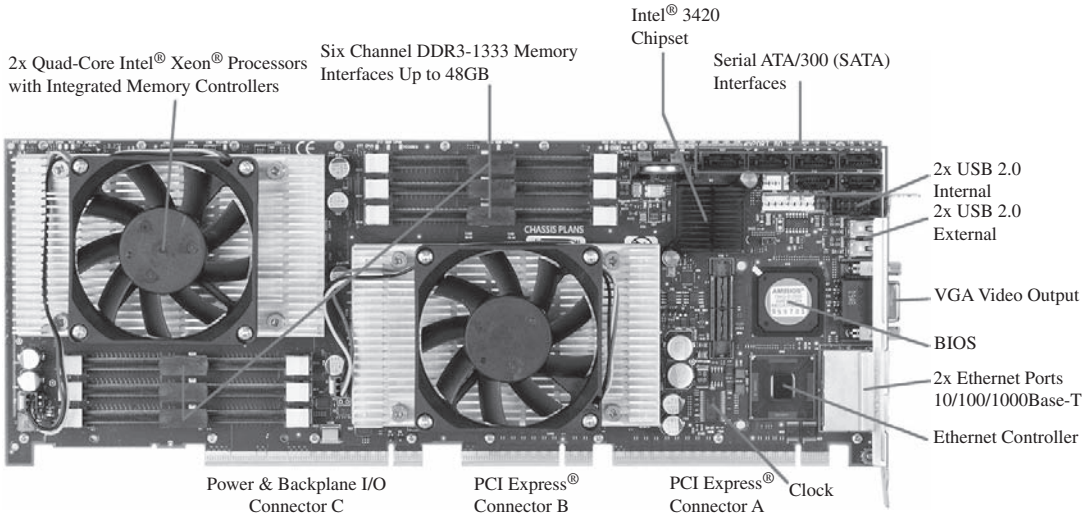


Figure 1.3 Motherboard with Two Intel Quad-Core Xeon Processors
Source: Chassis Plans, www.chassis-plans.com

- Serial ATA (SATA) sockets for connection to disk memory (Section 7.7 discusses Ethernet, USB, and SATA).
- Interfaces for DDR (double data rate) main memory chips (Section 5.3 discusses DDR).
- Intel 3420 chipset is an I/O controller for direct memory access operations between peripheral devices and main memory (Section 7.5 discusses DDR).

Following our top-down strategy, as illustrated in Figures 1.1 and 1.2, we can now zoom in and look at the internal structure of a processor chip. For variety, we look at an IBM chip instead of the Intel processor chip. Figure 1.4 is a photograph of the processor chip for the IBM zEnterprise EC12 mainframe computer. This chip has 2.75 billion transistors. The superimposed labels indicate how the silicon real estate of the chip is allocated. We see that this chip has six cores, or processors. In addition, there are two large areas labeled L3 cache, which are shared by all six processors. The L3 control logic controls traffic between the L3 cache and the cores and between the L3 cache and the external environment. Additionally, there is storage control (SC) logic between the cores and the L3 cache. The memory controller (MC) function controls access to memory external to the chip. The GX I/O bus controls the interface to the channel adapters accessing the I/O.

Going down one level deeper, we examine the internal structure of a single core, as shown in the photograph of Figure 1.5. Keep in mind that this is a portion of the silicon surface area making up a single-processor chip. The main sub-areas within this core area are the following:

- **ISU (instruction sequence unit):** Determines the sequence in which instructions are executed in what is referred to as a superscalar architecture (Chapter 16).
- **IFU (instruction fetch unit):** Logic for fetching instructions.

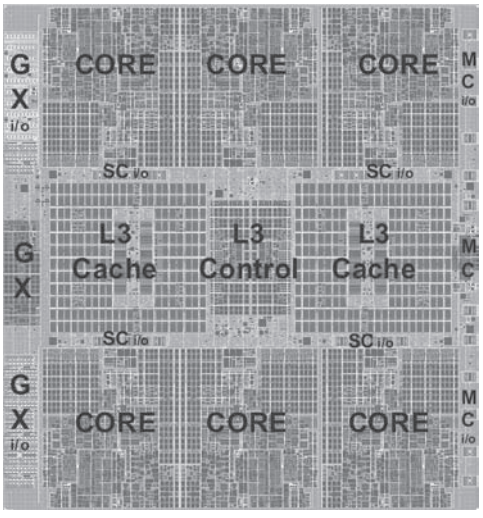


Figure 1.4 zEnterprise EC12 Processor Unit (PU) chip diagram
Source: IBM zEnterprise EC12 Technical Guide, December 2013, SG24-8049-01. IBM, Reprinted by Permission

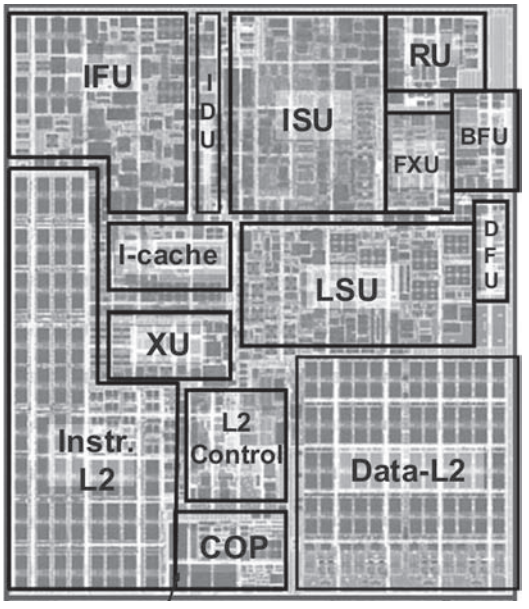


Figure 1.5 zEnterprise EC12 Core layout
Source: IBM zEnterprise EC12 Technical Guide, December 2013, SG24-8049-01. IBM, Reprinted by Permission

- **IDU (instruction decode unit):** The IDU is fed from the IFU buffers, and is responsible for the parsing and decoding of all z/Architecture operation codes.
- **LSU (load-store unit):** The LSU contains the 96-kB L1 data cache,[1] and manages data traffic between the L2 data cache and the functional execution units. It is responsible for handling all types of operand accesses of all lengths, modes, and formats as defined in the z/Architecture.
- **XU (translation unit):** This unit translates logical addresses from instructions into physical addresses in main memory. The XU also contains a translation lookaside buffer (TLB) used to speed up memory access. TLBs are discussed in Chapter 8.
- **FXU (fixed-point unit):** The FXU executes fixed-point arithmetic operations.
- **BFU (binary floating-point unit):** The BFU handles all binary and hexadecimal floating-point operations, as well as fixed-point multiplication operations.
- **DFU (decimal floating-point unit):** The DFU handles both fixed-point and floating-point operations on numbers that are stored as decimal digits.
- **RU (recovery unit):** The RU keeps a copy of the complete state of the system that includes all registers, collects hardware fault signals, and manages the hardware recovery actions.

[1]kB = kilobyte = 2048 bytes. Numerical prefixes are explained in a document under the "Other Useful" tab at ComputerScienceStudent.com.

- **COP (dedicated co-processor):** The COP is responsible for data compression and encryption functions for each core.
- **I-cache:** This is a 64-kB L1 instruction cache, allowing the IFU to prefetch instructions before they are needed.
- **L2 control:** This is the control logic that manages the traffic through the two L2 caches.
- **Data-L2:** A 1-MB L2 data cache for all memory traffic other than instructions.
- **Instr-L2:** A 1-MB L2 instruction cache.

As we progress through the book, the concepts introduced in this section will become clearer.

1.3 A BRIEF HISTORY OF COMPUTERS[2]

In this section, we provide a brief overview of the history of the development of computers. This history is interesting in itself, but more importantly, provides a basic introduction to many important concepts that we deal with throughout the book.

The First Generation: Vacuum Tubes

The first generation of computers used vacuum tubes for digital logic elements and memory. A number of research and then commercial computers were built using vacuum tubes. For our purposes, it will be instructive to examine perhaps the most famous first-generation computer, known as the IAS computer.

A fundamental design approach first implemented in the IAS computer is known as the *stored-program concept*. This idea is usually attributed to the mathematician John von Neumann. Alan Turing developed the idea at about the same time. The first publication of the idea was in a 1945 proposal by von Neumann for a new computer, the EDVAC (Electronic Discrete Variable Computer).[3]

In 1946, von Neumann and his colleagues began the design of a new stored-program computer, referred to as the IAS computer, at the Princeton Institute for Advanced Studies. The IAS computer, although not completed until 1952, is the prototype of all subsequent general-purpose computers.[4]

Figure 1.6 shows the structure of the IAS computer (compare with Figure 1.1). It consists of

- A **main memory**, which stores both data and instructions[5]
- An **arithmetic and logic unit (ALU)** capable of operating on binary data

---

[2]This book's Companion Web site (WilliamStallings.com/ComputerOrganization) contains several links to sites that provide photographs of many of the devices and components discussed in this section.

[3]The 1945 report on EDVAC is available at box.com/COA10e.

[4]A 1954 report [GOLD54] describes the implemented IAS machine and lists the final instruction set. It is available at box.com/COA10e.

[5]In this book, unless otherwise noted, the term *instruction* refers to a machine instruction that is directly interpreted and executed by the processor, in contrast to a statement in a high-level language, such as Ada or C++, which must first be compiled into a series of machine instructions before being executed.

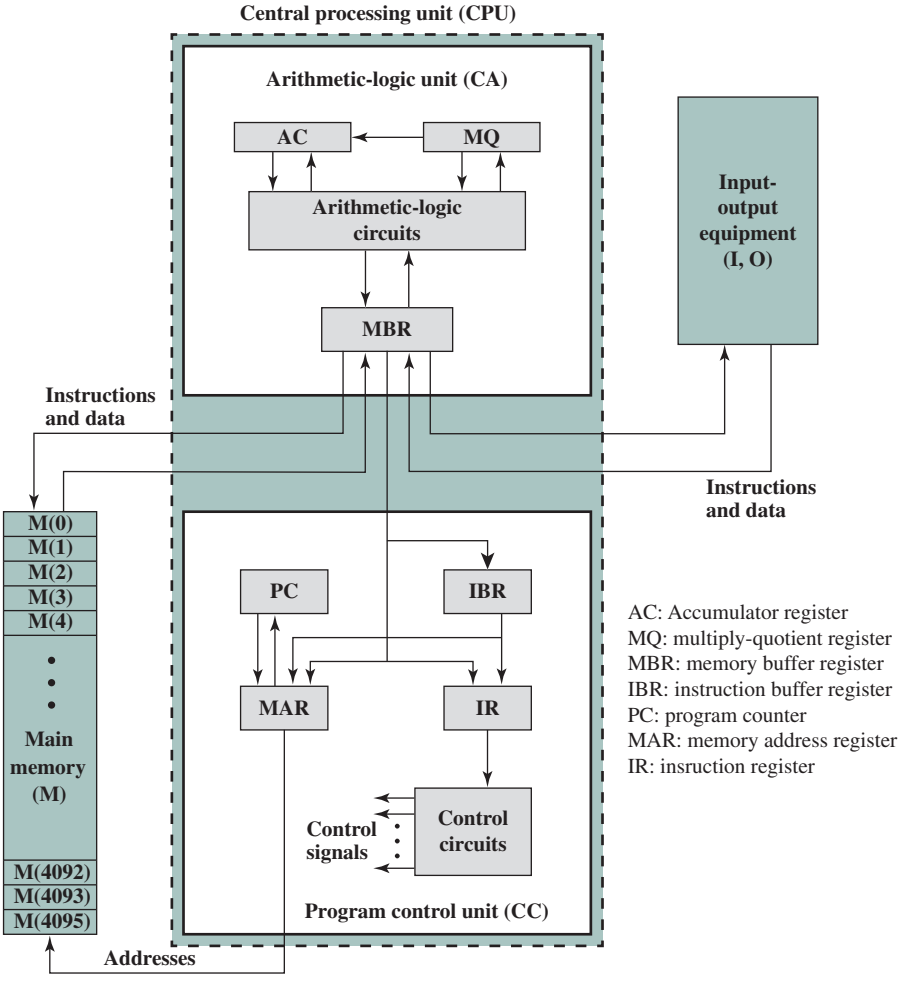


Figure 1.6 IAS Structure

- A **control unit**, which interprets the instructions in memory and causes them to be executed
- **Input–output (I/O)** equipment operated by the control unit

This structure was outlined in von Neumann's earlier proposal, which is worth quoting in part at this point [VONN45]:

> 2.2 **First:** Since the device is primarily a computer, it will have to perform the elementary operations of arithmetic most frequently. These are addition, subtraction, multiplication, and division. It is therefore reasonable that it should contain specialized organs for just these operations.

It must be observed, however, that while this principle as such is probably sound, the specific way in which it is realized requires close scrutiny. At any rate a *central arithmetical* part of the device will probably have to exist, and this constitutes *the first specific part: CA*.

2.3 **Second:** The logical control of the device, that is, the proper sequencing of its operations, can be most efficiently carried out by a central control organ. If the device is to be *elastic*, that is, as nearly as possible *all purpose*, then a distinction must be made between the specific instructions given for and defining a particular problem, and the general control organs that see to it that these instructions—no matter what they are—are carried out. The former must be stored in some way; the latter are represented by definite operating parts of the device. By the *central control* we mean this latter function only, and the organs that perform it form *the second specific part: CC*.

2.4 **Third:** Any device that is to carry out long and complicated sequences of operations (specifically of calculations) must have a considerable memory . . .

The instructions which govern a complicated problem may constitute considerable material, particularly so if the code is circumstantial (which it is in most arrangements). This material must be remembered.

At any rate, the total *memory* constitutes *the third specific part of the device: M.*

2.6 The three specific parts CA, CC (together C), and M correspond to the *associative* neurons in the human nervous system. It remains to discuss the equivalents of the *sensory* or *afferent* and the *motor* or *efferent* neurons. These are the *input* and *output* organs of the device.

The device must be endowed with the ability to maintain input and output (sensory and motor) contact with some specific medium of this type. The medium will be called the *outside recording medium of the device: R*.

2.7 **Fourth:** The device must have organs to transfer information from R into its specific parts C and M. These organs form its *input*, the *fourth specific part: I*. It will be seen that it is best to make all transfers from R (by I) into M and never directly from C.

2.8 **Fifth:** The device must have organs to transfer from its specific parts C and M into R. These organs form its *output*, *the fifth specific part: O*. It will be seen that it is again best to make all transfers from M (by O) into R, and never directly from C.

With rare exceptions, all of today's computers have this same general structure and function and are thus referred to as *von Neumann machines*. Thus, it is worthwhile at this point to describe briefly the operation of the IAS computer [BURK46, GOLD54]. Following [HAYE98], the terminology and notation of von Neumann

are changed in the following to conform more closely to modern usage; the examples accompanying this discussion are based on that latter text.

The memory of the IAS consists of 4,096 storage locations, called *words*, of 40 binary digits (bits) each.[6] Both data and instructions are stored there. Numbers are represented in binary form, and each instruction is a binary code. Figure 1.7 illustrates these formats. Each number is represented by a sign bit and a 39-bit value. A word may alternatively contain two 20-bit instructions, with each instruction consisting of an 8-bit operation code (opcode) specifying the operation to be performed and a 12-bit address designating one of the words in memory (numbered from 0 to 999).

The control unit operates the IAS by fetching instructions from memory and executing them one at a time. We explain these operations with reference to Figure 1.6. This figure reveals that both the control unit and the ALU contain storage locations, called *registers*, defined as follows:

- **Memory buffer register (MBR):** Contains a word to be stored in memory or sent to the I/O unit, or is used to receive a word from memory or from the I/O unit.
- **Memory address register (MAR):** Specifies the address in memory of the word to be written from or read into the MBR.
- **Instruction register (IR):** Contains the 8-bit opcode instruction being executed.
- **Instruction buffer register (IBR):** Employed to hold temporarily the right-hand instruction from a word in memory.
- **Program counter (PC):** Contains the address of the next instruction pair to be fetched from memory.
- **Accumulator (AC) and multiplier quotient (MQ):** Employed to hold temporarily operands and results of ALU operations. For example, the result

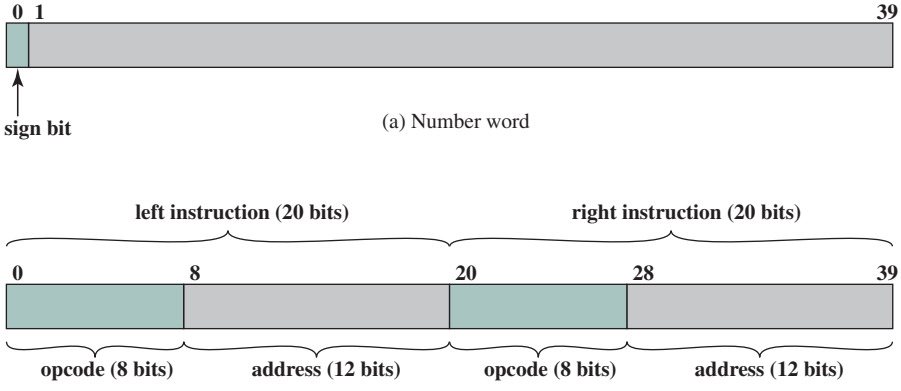


Figure 1.7 IAS Memory Formats

[6]There is no universal definition of the term *word*. In general, a word is an ordered set of bytes or bits that is the normal unit in which information may be stored, transmitted, or operated on within a given computer. Typically, if a processor has a fixed-length instruction set, then the instruction length equals the word length.

of multiplying two 40-bit numbers is an 80-bit number; the most significant 40 bits are stored in the AC and the least significant in the MQ.

The IAS operates by repetitively performing an *instruction cycle*, as shown in Figure 1.8. Each instruction cycle consists of two subcycles. During the *fetch cycle*, the opcode of the next instruction is loaded into the IR and the address portion is loaded into the MAR. This instruction may be taken from the IBR, or it can be obtained from memory by loading a word into the MBR, and then down to the IBR, IR, and MAR.

Why the indirection? These operations are controlled by electronic circuitry and result in the use of data paths. To simplify the electronics, there is only one register that is used to specify the address in memory for a read or write and only one register used for the source or destination.

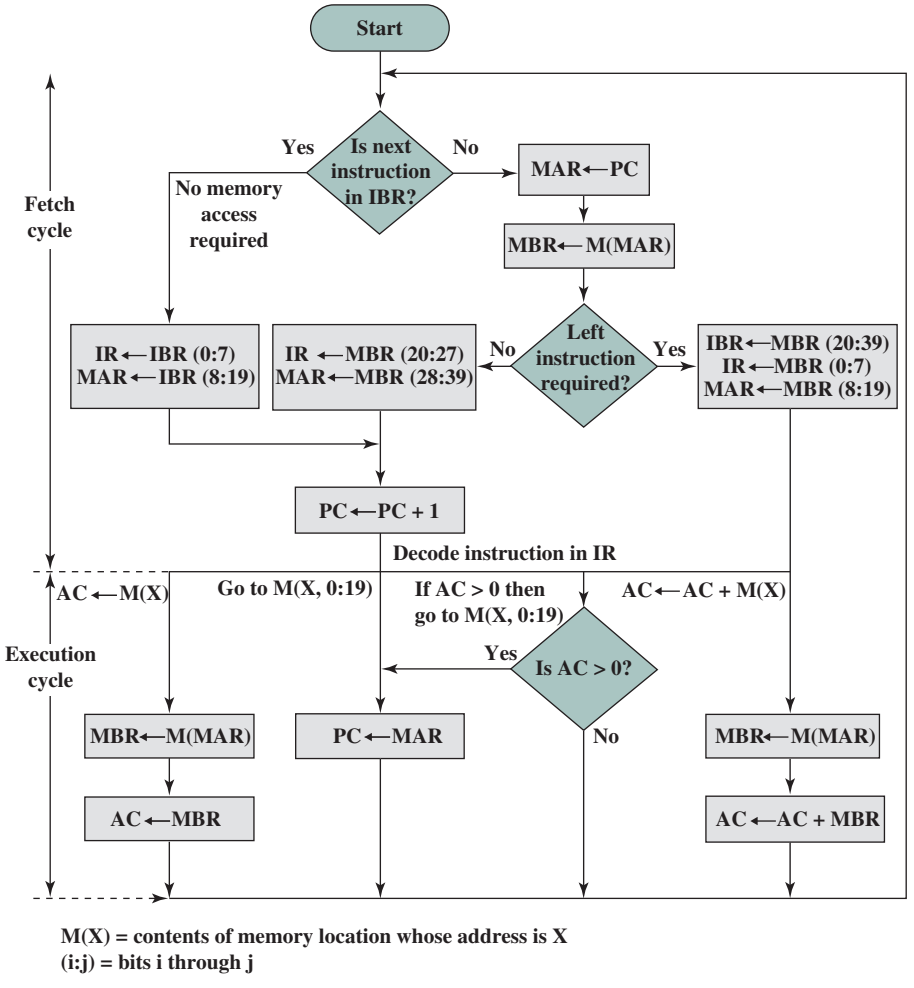


Figure 1.8 Partial Flowchart of IAS Operation

Once the opcode is in the IR, the *execute cycle* is performed. Control circuitry interprets the opcode and executes the instruction by sending out the appropriate control signals to cause data to be moved or an operation to be performed by the ALU.

The IAS computer had a total of 21 instructions, which are listed in Table 1.1. These can be grouped as follows:

- **Data transfer:** Move data between memory and ALU registers or between two ALU registers.
- **Unconditional branch:** Normally, the control unit executes instructions in sequence from memory. This sequence can be changed by a branch instruction, which facilitates repetitive operations.

Table 1.1 The IAS Instruction Set

Instruction Type	Opcode	Symbolic Representation	Description
Data transfer	00001010	LOAD MQ	Transfer contents of register MQ to the accumulator AC
	00001001	LOAD MQ,M(X)	Transfer contents of memory location X to MQ
	00100001	STOR M(X)	Transfer contents of accumulator to memory location X
	00000001	LOAD M(X)	Transfer M(X) to the accumulator
	00000010	LOAD –M(X)	Transfer –M(X) to the accumulator
	00000011	LOAD |M(X)|	Transfer absolute value of M(X) to the accumulator
	00000100	LOAD –|M(X)|	Transfer –|M(X)| to the accumulator
Unconditional branch	00001101	JUMP M(X,0:19)	Take next instruction from left half of M(X)
	00001110	JUMP M(X,20:39)	Take next instruction from right half of M(X)
Conditional branch	00001111	JUMP + M(X,0:19)	If number in the accumulator is nonnegative, take next instruction from left half of M(X)
	00010000	JUMP + M(X,20:39)	If number in the accumulator is nonnegative, take next instruction from right half of M(X)
Arithmetic	00000101	ADD M(X)	Add M(X) to AC; put the result in AC
	00000111	ADD |M(X)|	Add |M(X)| to AC; put the result in AC
	00000110	SUB M(X)	Subtract M(X) from AC; put the result in AC
	00001000	SUB |M(X)|	Subtract |M(X)| from AC; put the remainder in AC
	00001011	MUL M(X)	Multiply M(X) by MQ; put most significant bits of result in AC, put least significant bits in MQ
	00001100	DIV M(X)	Divide AC by M(X); put the quotient in MQ and the remainder in AC
	00010100	LSH	Multiply accumulator by 2; that is, shift left one bit position
	00010101	RSH	Divide accumulator by 2; that is, shift right one position
Address modify	00010010	STOR M(X,8:19)	Replace left address field at M(X) by 12 rightmost bits of AC
	00010011	STOR M(X,28:39)	Replace right address field at M(X) by 12 rightmost bits of AC

- **Conditional branch:** The branch can be made dependent on a condition, thus allowing decision points.
- **Arithmetic:** Operations performed by the ALU.
- **Address modify:** Permits addresses to be computed in the ALU and then inserted into instructions stored in memory. This allows a program considerable addressing flexibility.

Table 1.1 presents instructions (excluding I/O instructions) in a symbolic, easy-to-read form. In binary form, each instruction must conform to the format of Figure 1.7b. The opcode portion (first 8 bits) specifies which of the 21 instructions is to be executed. The address portion (remaining 12 bits) specifies which of the 4,096 memory locations is to be involved in the execution of the instruction.

Figure 1.8 shows several examples of instruction execution by the control unit. Note that each operation requires several steps, some of which are quite elaborate. The multiplication operation requires 39 suboperations, one for each bit position except that of the sign bit.

The Second Generation: Transistors

The first major change in the electronic computer came with the replacement of the vacuum tube by the transistor. The transistor, which is smaller, cheaper, and generates less heat than a vacuum tube, can be used in the same way as a vacuum tube to construct computers. Unlike the vacuum tube, which requires wires, metal plates, a glass capsule, and a vacuum, the transistor is a *solid-state device*, made from silicon.

The transistor was invented at Bell Labs in 1947 and by the 1950s had launched an electronic revolution. It was not until the late 1950s, however, that fully transistorized computers were commercially available. The use of the transistor defines the *second generation* of computers. It has become widely accepted to classify computers into generations based on the fundamental hardware technology employed (Table 1.2). Each new generation is characterized by greater processing performance, larger memory capacity, and smaller size than the previous one.

But there are other changes as well. The second generation saw the introduction of more complex arithmetic and logic units and control units, the use of high-level programming languages, and the provision of *system software* with the

Table 1.2 Computer Generations

Generation	Approximate Dates	Technology	Typical Speed (operations per second)
1	1946–1957	Vacuum tube	40,000
2	1957–1964	Transistor	200,000
3	1965–1971	Small- and medium-scale integration	1,000,000
4	1972–1977	Large scale integration	10,000,000
5	1978–1991	Very large scale integration	100,000,000
6	1991–	Ultra large scale integration	>1,000,000,000

computer. In broad terms, system software provided the ability to load programs, move data to peripherals, and libraries to perform common computations, similar to what modern operating systems, such as Windows and Linux, do.

It will be useful to examine an important member of the second generation: the IBM 7094 [BELL71]. From the introduction of the 700 series in 1952 to the introduction of the last member of the 7000 series in 1964, this IBM product line underwent an evolution that is typical of computer products. Successive members of the product line showed increased performance, increased capacity, and/or lower cost.

The size of main memory, in multiples of 2^{10} 36-bit words, grew from 2k ($1k = 2^{10}$) to 32k words,[7] while the time to access one word of memory, the *memory cycle time*, fell from 30 μs to 1.4 μs. The number of opcodes grew from a modest 24 to 185.

Also, over the lifetime of this series of computers, the relative speed of the CPU increased by a factor of 50. Speed improvements are achieved by improved electronics (e.g., a transistor implementation is faster than a vacuum tube implementation) and more complex circuitry. For example, the IBM 7094 includes an Instruction Backup Register, used to buffer the next instruction. The control unit fetches two adjacent words from memory for an instruction fetch. Except for the occurrence of a branching instruction, which is relatively infrequent (perhaps 10 to 15%), this means that the control unit has to access memory for an instruction on only half the instruction cycles. This prefetching significantly reduces the average instruction cycle time.

Figure 1.9 shows a large (many peripherals) configuration for an IBM 7094, which is representative of second-generation computers. Several differences from the IAS computer are worth noting. The most important of these is the use of *data channels*. A data channel is an independent I/O module with its own processor and instruction set. In a computer system with such devices, the CPU does not execute detailed I/O instructions. Such instructions are stored in a main memory to be executed by a special-purpose processor in the data channel itself. The CPU initiates an I/O transfer by sending a control signal to the data channel, instructing it to execute a sequence of instructions in memory. The data channel performs its task independently of the CPU and signals the CPU when the operation is complete. This arrangement relieves the CPU of a considerable processing burden.

Another new feature is the *multiplexor*, which is the central termination point for data channels, the CPU, and memory. The multiplexor schedules access to the memory from the CPU and data channels, allowing these devices to act independently.

The Third Generation: Integrated Circuits

A single, self-contained transistor is called a *discrete component*. Throughout the 1950s and early 1960s, electronic equipment was composed largely of discrete components—transistors, resistors, capacitors, and so on. Discrete components were manufactured separately, packaged in their own containers, and soldered or wired

[7] A discussion of the uses of numerical prefixes, such as kilo and giga, is contained in a supporting document at the Computer Science Student Resource Site at ComputerScienceStudent.com.

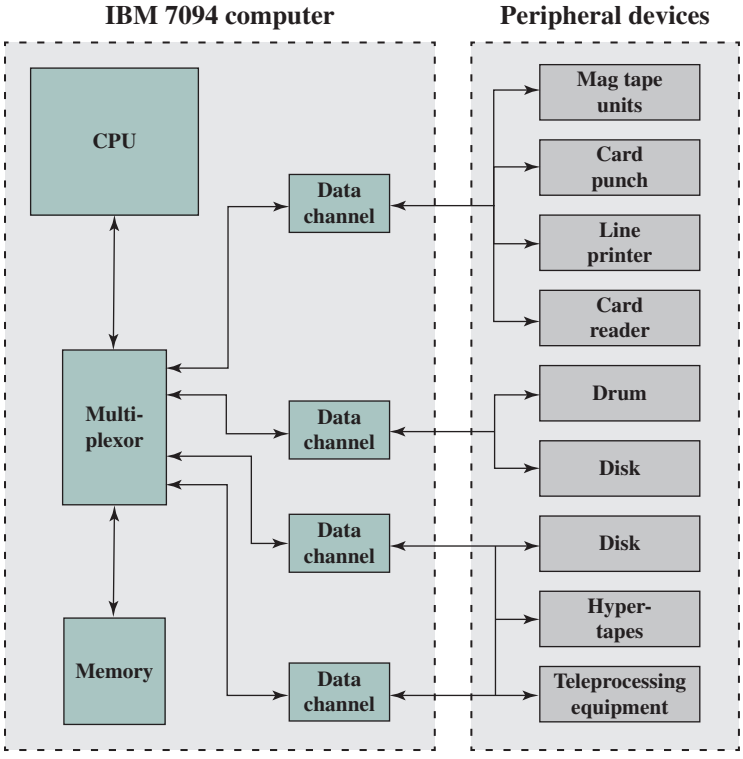


Figure 1.9 An IBM 7094 Configuration

together onto Masonite-like circuit boards, which were then installed in computers, oscilloscopes, and other electronic equipment. Whenever an electronic device called for a transistor, a little tube of metal containing a pinhead-sized piece of silicon had to be soldered to a circuit board. The entire manufacturing process, from transistor to circuit board, was expensive and cumbersome.

These facts of life were beginning to create problems in the computer industry. Early second-generation computers contained about 10,000 transistors. This figure grew to the hundreds of thousands, making the manufacture of newer, more powerful machines increasingly difficult.

In 1958 came the achievement that revolutionized electronics and started the era of microelectronics: the invention of the integrated circuit. It is the integrated circuit that defines the third generation of computers. In this section, we provide a brief introduction to the technology of integrated circuits. Then we look at perhaps the two most important members of the third generation, both of which were introduced at the beginning of that era: the IBM System/360 and the DEC PDP-8.

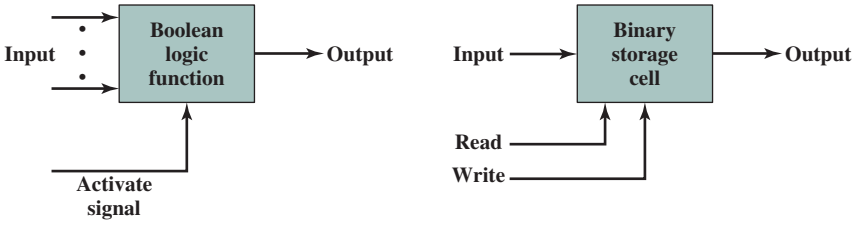
MICROELECTRONICS Microelectronics means, literally, "small electronics." Since the beginnings of digital electronics and the computer industry, there has been a persistent and consistent trend toward the reduction in size of digital electronic circuits. Before examining the implications and benefits of this trend, we need to say something about the nature of digital electronics. A more detailed discussion is found in Chapter 11.

The basic elements of a digital computer, as we know, must perform data storage, movement, processing, and control functions. Only two fundamental types of components are required (Figure 1.10): gates and memory cells. A **gate** is a device that implements a simple Boolean or logical function. For example, an AND gate with inputs *A* and *B* and output *C* implements the expression IF *A* AND *B* ARE TRUE THEN *C* IS TRUE. Such devices are called gates because they control data flow in much the same way that canal gates control the flow of water. The **memory cell** is a device that can store 1 bit of data; that is, the device can be in one of two stable states at any time. By interconnecting large numbers of these fundamental devices, we can construct a computer. We can relate this to our four basic functions as follows:

- **Data storage:** Provided by memory cells.
- **Data processing:** Provided by gates.
- **Data movement:** The paths among components are used to move data from memory to memory and from memory through gates to memory.
- **Control:** The paths among components can carry control signals. For example, a gate will have one or two data inputs plus a control signal input that activates the gate. When the control signal is ON, the gate performs its function on the data inputs and produces a data output. Conversely, when the control signal is OFF, the output line is null, such as the one produced by a high impedance state. Similarly, the memory cell will store the bit that is on its input lead when the WRITE control signal is ON and will place the bit that is in the cell on its output lead when the READ control signal is ON.

Thus, a computer consists of gates, memory cells, and interconnections among these elements. The gates and memory cells are, in turn, constructed of simple electronic components, such as transistors and capacitors.

The integrated circuit exploits the fact that such components as transistors, resistors, and conductors can be fabricated from a semiconductor such as silicon. It is merely an extension of the solid-state art to fabricate an entire circuit in a tiny piece of silicon rather than assemble discrete components made from separate pieces of silicon into the same circuit. Many transistors can be produced at the same time on a single wafer of silicon. Equally important, these transistors can be connected with a process of metallization to form circuits.



(a) Gate (b) Memory cell

Figure 1.10 Fundamental Computer Elements

Figure 1.11 depicts the key concepts in an integrated circuit. A thin *wafer* of silicon is divided into a matrix of small areas, each a few millimeters square. The identical circuit pattern is fabricated in each area, and the wafer is broken up into *chips*. Each chip consists of many gates and/or memory cells plus a number of input and output attachment points. This chip is then packaged in housing that protects it and provides pins for attachment to devices beyond the chip. A number of these packages can then be interconnected on a printed circuit board to produce larger and more complex circuits.

Initially, only a few gates or memory cells could be reliably manufactured and packaged together. These early integrated circuits are referred to as **small-scale integration (SSI)**. As time went on, it became possible to pack more and more components on the same chip. This growth in density is illustrated in Figure 1.12; it is one of the most remarkable technological trends ever recorded.[8] This figure reflects the famous Moore's law, which was propounded by Gordon Moore, cofounder of Intel, in 1965 [MOOR65]. Moore observed that the number of transistors that could be put on a single chip was doubling every year, and correctly predicted that this pace would continue into the near future. To the surprise of many, including Moore, the pace continued year after year and decade after decade. The pace slowed to a doubling every 18 months in the 1970s but has sustained that rate ever since.

The consequences of Moore's law are profound:

1. The cost of a chip has remained virtually unchanged during this period of rapid growth in density. This means that the cost of computer logic and memory circuitry has fallen at a dramatic rate.

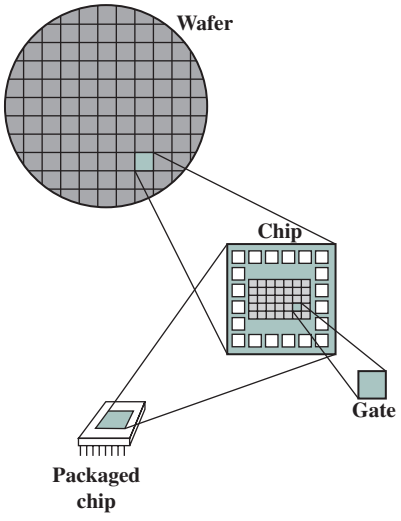


Figure 1.11 Relationship among Wafer, Chip, and Gate

[8]Note that the vertical axis uses a log scale. A basic review of log scales is in the math refresher document at the Computer Science Student Resource Site at ComputerScienceStudent.com.

2. Because logic and memory elements are placed closer together on more densely packed chips, the electrical path length is shortened, increasing operating speed.
3. The computer becomes smaller, making it more convenient to place in a variety of environments.
4. There is a reduction in power requirements.
5. The interconnections on the integrated circuit are much more reliable than solder connections. With more circuitry on each chip, there are fewer interchip connections.

IBM SYSTEM/360 By 1964, IBM had a firm grip on the computer market with its 7000 series of machines. In that year, IBM announced the System/360, a new family of computer products. Although the announcement itself was no surprise, it contained some unpleasant news for current IBM customers: the 360 product line was incompatible with older IBM machines. Thus, the transition to the 360 would be difficult for the current customer base, but IBM felt this was necessary to break out of some of the constraints of the 7000 architecture and to produce a system capable of evolving with the new integrated circuit technology [PADE81, GIFF87]. The strategy paid off both financially and technically. The 360 was the success of the decade and cemented IBM as the overwhelmingly dominant computer vendor, with a market share above 70%. And, with some modifications and extensions, the architecture of the 360 remains to this day the architecture of IBM's mainframe[9] computers. Examples using this architecture can be found throughout this text.

The System/360 was the industry's first planned family of computers. The family covered a wide range of performance and cost. The models were compatible in the

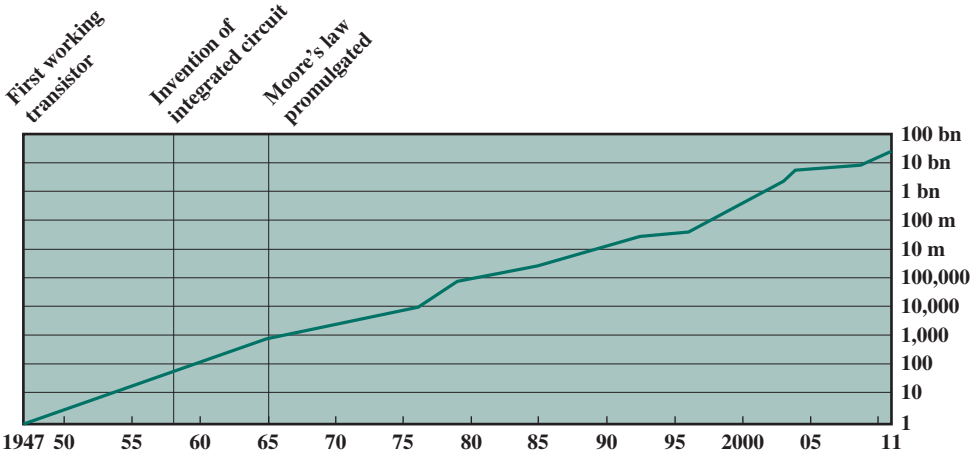


Figure 1.12 Growth in Transistor Count on Integrated Circuits

[9]The term *mainframe* is used for the larger, most powerful computers other than supercomputers. Typical characteristics of a mainframe are that it supports a large database, has elaborate I/O hardware, and is used in a central data processing facility.

sense that a program written for one model should be capable of being executed by another model in the series, with only a difference in the time it takes to execute.

The concept of a family of compatible computers was both novel and extremely successful. A customer with modest requirements and a budget to match could start with the relatively inexpensive Model 30. Later, if the customer's needs grew, it was possible to upgrade to a faster machine with more memory without sacrificing the investment in already-developed software. The characteristics of a family are as follows:

- **Similar or identical instruction set:** In many cases, the exact same set of machine instructions is supported on all members of the family. Thus, a program that executes on one machine will also execute on any other. In some cases, the lower end of the family has an instruction set that is a subset of that of the top end of the family. This means that programs can move up but not down.
- **Similar or identical operating system:** The same basic operating system is available for all family members. In some cases, additional features are added to the higher-end members.
- **Increasing speed:** The rate of instruction execution increases in going from lower to higher family members.
- **Increasing number of I/O ports:** The number of I/O ports increases in going from lower to higher family members.
- **Increasing memory size:** The size of main memory increases in going from lower to higher family members.
- **Increasing cost:** At a given point in time, the cost of a system increases in going from lower to higher family members.

How could such a family concept be implemented? Differences were achieved based on three factors: basic speed, size, and degree of simultaneity [STEV64]. For example, greater speed in the execution of a given instruction could be gained by the use of more complex circuitry in the ALU, allowing suboperations to be carried out in parallel. Another way of increasing speed was to increase the width of the data path between main memory and the CPU. On the Model 30, only 1 byte (8 bits) could be fetched from main memory at a time, whereas 8 bytes could be fetched at a time on the Model 75.

The System/360 not only dictated the future course of IBM but also had a profound impact on the entire industry. Many of its features have become standard on other large computers.

DEC PDP-8 In the same year that IBM shipped its first System/360, another momentous first shipment occurred: PDP-8 from Digital Equipment Corporation (DEC). At a time when the average computer required an air-conditioned room, the PDP-8 (dubbed a minicomputer by the industry, after the miniskirt of the day) was small enough that it could be placed on top of a lab bench or be built into other equipment. It could not do everything the mainframe could, but at $16,000, it was cheap enough for each lab technician to have one. In contrast, the System/360 series of mainframe computers introduced just a few months before cost hundreds of thousands of dollars.

The low cost and small size of the PDP-8 enabled another manufacturer to purchase a PDP-8 and integrate it into a total system for resale. These other manufacturers came to be known as **original equipment manufacturers (OEMs)**, and the OEM market became and remains a major segment of the computer marketplace.

In contrast to the central-switched architecture (Figure 1.9) used by IBM on its 700/7000 and 360 systems, later models of the PDP-8 used a structure that became virtually universal for microcomputers: the bus structure. This is illustrated in Figure 1.13. The PDP-8 bus, called the Omnibus, consists of 96 separate signal paths, used to carry control, address, and data signals. Because all system components share a common set of signal paths, their use can be controlled by the CPU. This architecture is highly flexible, allowing modules to be plugged into the bus to create various configurations. It is only in recent years that the bus structure has given way to a structure known as point-to-point interconnect, described in Chapter 3.

Later Generations

Beyond the third generation there is less general agreement on defining generations of computers. Table 1.2 suggests that there have been a number of later generations, based on advances in integrated circuit technology. With the introduction of **large-scale integration (LSI)**, more than 1,000 components can be placed on a single integrated circuit chip. Very-large-scale integration (VLSI) achieved more than 10,000 components per chip, while current ultra-large-scale integration (ULSI) chips can contain more than one billion components.

With the rapid pace of technology, the high rate of introduction of new products, and the importance of software and communications as well as hardware, the classification by generation becomes less clear and less meaningful. In this section, we mention two of the most important of developments in later generations.

SEMICONDUCTOR MEMORY The first application of integrated circuit technology to computers was the construction of the processor (the control unit and the arithmetic and logic unit) out of integrated circuit chips. But it was also found that this same technology could be used to construct memories.

In the 1950s and 1960s, most computer memory was constructed from tiny rings of ferromagnetic material, each about a sixteenth of an inch in diameter. These rings were strung up on grids of fine wires suspended on small screens inside the computer. Magnetized one way, a ring (called a *core*) represented a one; magnetized the other way, it stood for a zero. Magnetic-core memory was rather fast; it took as little as a millionth of a second to read a bit stored in memory. But it was

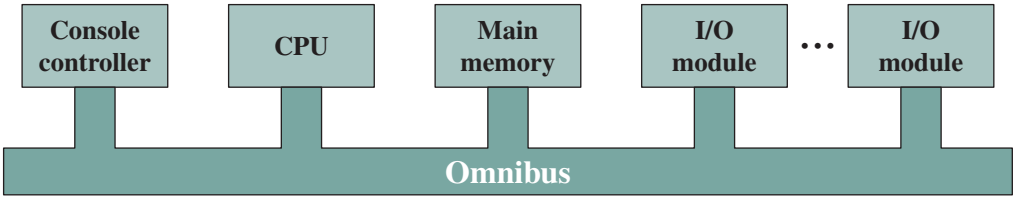


Figure 1.13 PDP-8 Bus Structure

expensive and bulky, and used destructive readout: The simple act of reading a core erased the data stored in it. It was therefore necessary to install circuits to restore the data as soon as it had been extracted.

Then, in 1970, Fairchild produced the first relatively capacious semiconductor memory. This chip, about the size of a single core, could hold 256 bits of memory. It was nondestructive and much faster than core. It took only 70 billionths of a second to read a bit. However, the cost per bit was higher than for that of core.

In 1974, a seminal event occurred: The price per bit of semiconductor memory dropped below the price per bit of core memory. Following this, there has been a continuing and rapid decline in memory cost accompanied by a corresponding increase in physical memory density. This has led the way to smaller, faster machines with memory sizes of larger and more expensive machines from just a few years earlier. Developments in memory technology, together with developments in processor technology to be discussed next, changed the nature of computers in less than a decade. Although bulky, expensive computers remain a part of the landscape, the computer has also been brought out to the "end user," with office machines and personal computers.

Since 1970, semiconductor memory has been through 13 generations: 1k, 4k, 16k, 64k, 256k, 1M, 4M, 16M, 64M, 256M, 1G, 4G, and, as of this writing, 8 Gb on a single chip ($1\,\text{k} = 2^{10}, 1\,\text{M} = 2^{20}, 1\,\text{G} = 2^{30}$). Each generation has provided increased storage density, accompanied by declining cost per bit and declining access time. Densities are projected to reach 16 Gb by 2018 and 32 Gb by 2023 [ITRS14].

MICROPROCESSORS Just as the density of elements on memory chips has continued to rise, so has the density of elements on processor chips. As time went on, more and more elements were placed on each chip, so that fewer and fewer chips were needed to construct a single computer processor.

A breakthrough was achieved in 1971, when Intel developed its 4004. The 4004 was the first chip to contain *all* of the components of a CPU on a single chip: The microprocessor was born.

The 4004 can add two 4-bit numbers and can multiply only by repeated addition. By today's standards, the 4004 is hopelessly primitive, but it marked the beginning of a continuing evolution of microprocessor capability and power.

This evolution can be seen most easily in the number of bits that the processor deals with at a time. There is no clear-cut measure of this, but perhaps the best measure is the data bus width: the number of bits of data that can be brought into or sent out of the processor at a time. Another measure is the number of bits in the accumulator or in the set of general-purpose registers. Often, these measures coincide, but not always. For example, a number of microprocessors were developed that operate on 16-bit numbers in registers but can only read and write 8 bits at a time.

The next major step in the evolution of the microprocessor was the introduction in 1972 of the Intel 8008. This was the first 8-bit microprocessor and was almost twice as complex as the 4004.

Neither of these steps was to have the impact of the next major event: the introduction in 1974 of the Intel 8080. This was the first general-purpose microprocessor. Whereas the 4004 and the 8008 had been designed for specific applications, the 8080 was designed to be the CPU of a general-purpose microcomputer. Like the

8008, the 8080 is an 8-bit microprocessor. The 8080, however, is faster, has a richer instruction set, and has a large addressing capability.

About the same time, 16-bit microprocessors began to be developed. However, it was not until the end of the 1970s that powerful, general-purpose 16-bit microprocessors appeared. One of these was the 8086. The next step in this trend occurred in 1981, when both Bell Labs and Hewlett-Packard developed 32-bit, single-chip microprocessors. Intel introduced its own 32-bit microprocessor, the 80386, in 1985 (Table 1.3).

Table 1.3 Evolution of Intel Microprocessors (page 1 of 2)

(a) 1970s Processors

	4004	8008	8080	8086	8088
Introduced	1971	1972	1974	1978	1979
Clock speeds	108 kHz	108 kHz	2 MHz	5 MHz, 8 MHz, 10 MHz	5 MHz, 8 MHz
Bus width	4 bits	8 bits	8 bits	16 bits	8 bits
Number of transistors	2,300	3,500	6,000	29,000	29,000
Feature size (μm)	10	8	6	3	6
Addressable memory	640 bytes	16 KB	64 KB	1 MB	1 MB

(b) 1980s Processors

	80286	386TM DX	386TM SX	486TM DX CPU
Introduced	1982	1985	1988	1989
Clock speeds	6–12.5 MHz	16–33 MHz	16–33 MHz	25–50 MHz
Bus width	16 bits	32 bits	16 bits	32 bits
Number of transistors	134,000	275,000	275,000	1.2 million
Feature size (μm)	1.5	1	1	0.8–1
Addressable memory	16 MB	4 GB	16 MB	4 GB
Virtual memory	1 GB	64 TB	64 TB	64 TB
Cache	—	—	—	8 kB

(c) 1990s Processors

	486TM SX	Pentium	Pentium Pro	Pentium II
Introduced	1991	1993	1995	1997
Clock speeds	16–33 MHz	60–166 MHz,	150–200 MHz	200–300 MHz
Bus width	32 bits	32 bits	64 bits	64 bits
Number of transistors	1.185 million	3.1 million	5.5 million	7.5 million
Feature size (μm)	1	0.8	0.6	0.35
Addressable memory	4 GB	4 GB	64 GB	64 GB
Virtual memory	64 TB	64 TB	64 TB	64 TB
Cache	8 kB	8 kB	512 kB L1 and 1 MB L2	512 kB L2

(d) Recent Processors

	Pentium III	Pentium 4	Core 2 Duo	Core i7 EE 4960X
Introduced	1999	2000	2006	2013
Clock speeds	450–660 MHz	1.3–1.8 GHz	1.06–1.2 GHz	4 GHz
Bus width	64 bits	64 bits	64 bits	64 bits
Number of transistors	9.5 million	42 million	167 million	1.86 billion
Feature size (nm)	250	180	65	22
Addressable memory	64 GB	64 GB	64 GB	64 GB
Virtual memory	64 TB	64 TB	64 TB	64 TB
Cache	512 kB L2	256 kB L2	2 MB L2	1.5 MB L2/15 MB L3
Number of cores	1	1	2	6

1.4 THE EVOLUTION OF THE INTEL x86 ARCHITECTURE

Throughout this book, we rely on many concrete examples of computer design and implementation to illustrate concepts and to illuminate trade-offs. Numerous systems, both contemporary and historical, provide examples of important computer architecture design features. But the book relies principally on examples from two processor families: the Intel x86 and the ARM architectures. The current x86 offerings represent the results of decades of design effort on **complex instruction set computers (CISCs)**. The x86 incorporates the sophisticated design principles once found only on mainframes and supercomputers and serves as an excellent example of CISC design. An alternative approach to processor design is the **reduced instruction set computer (RISC)**. The ARM architecture is used in a wide variety of embedded systems and is one of the most powerful and best-designed RISC-based systems on the market. In this section and the next, we provide a brief overview of these two systems.

In terms of market share, Intel has ranked as the number one maker of microprocessors for non-embedded systems for decades, a position it seems unlikely to yield. The evolution of its flagship microprocessor product serves as a good indicator of the evolution of computer technology in general.

Table 1.3 shows that evolution. Interestingly, as microprocessors have grown faster and much more complex, Intel has actually picked up the pace. Intel used to develop microprocessors one after another, every four years. But Intel hopes to keep rivals at bay by trimming a year or two off this development time, and has done so with the most recent x86 generations.[10]

[10]Intel refers to this as the *tick-tock model*. Using this model, Intel has successfully delivered next-generation silicon technology as well as new processor microarchitecture on alternating years for the past several years. See http://www.intel.com/content/www/us/en/silicon-innovations/intel-tick-tock-model-general.html.

It is worthwhile to list some of the highlights of the evolution of the Intel product line:

- **8080:** The world's first general-purpose microprocessor. This was an 8-bit machine, with an 8-bit data path to memory. The 8080 was used in the first personal computer, the Altair.
- **8086:** A far more powerful, 16-bit machine. In addition to a wider data path and larger registers, the 8086 sported an instruction cache, or queue, that prefetches a few instructions before they are executed. A variant of this processor, the 8088, was used in IBM's first personal computer, securing the success of Intel. The 8086 is the first appearance of the x86 architecture.
- **80286:** This extension of the 8086 enabled addressing a 16-MB memory instead of just 1 MB.
- **80386:** Intel's first 32-bit machine, and a major overhaul of the product. With a 32-bit architecture, the 80386 rivaled the complexity and power of minicomputers and mainframes introduced just a few years earlier. This was the first Intel processor to support multitasking, meaning it could run multiple programs at the same time.
- **80486:** The 80486 introduced the use of much more sophisticated and powerful cache technology and sophisticated instruction pipelining. The 80486 also offered a built-in math coprocessor, offloading complex math operations from the main CPU.
- **Pentium:** With the Pentium, Intel introduced the use of superscalar techniques, which allow multiple instructions to execute in parallel.
- **Pentium Pro:** The Pentium Pro continued the move into superscalar organization begun with the Pentium, with aggressive use of register renaming, branch prediction, data flow analysis, and speculative execution.
- **Pentium II:** The Pentium II incorporated Intel MMX technology, which is designed specifically to process video, audio, and graphics data efficiently.
- **Pentium III:** The Pentium III incorporates additional floating-point instructions: The Streaming SIMD Extensions (SSE) instruction set extension added 70 new instructions designed to increase performance when exactly the same operations are to be performed on multiple data objects. Typical applications are digital signal processing and graphics processing.
- **Pentium 4:** The Pentium 4 includes additional floating-point and other enhancements for multimedia.[11]
- **Core:** This is the first Intel x86 microprocessor with a dual core, referring to the implementation of two cores on a single chip.
- **Core 2:** The Core 2 extends the Core architecture to 64 bits. The Core 2 Quad provides four cores on a single chip. More recent Core offerings have up to 10 cores per chip. An important addition to the architecture was the Advanced Vector Extensions instruction set that provided a set of 256-bit, and then 512-bit, instructions for efficient processing of vector data.

[11]With the Pentium 4, Intel switched from Roman numerals to Arabic numerals for model numbers.

Almost 40 years after its introduction in 1978, the x86 architecture continues to dominate the processor market outside of embedded systems. Although the organization and technology of the x86 machines have changed dramatically over the decades, the instruction set architecture has evolved to remain backward compatible with earlier versions. Thus, any program written on an older version of the x86 architecture can execute on newer versions. All changes to the instruction set architecture have involved additions to the instruction set, with no subtractions. The rate of change has been the addition of roughly one instruction per month added to the architecture [ANTH08], so that there are now thousands of instructions in the instruction set.

The x86 provides an excellent illustration of the advances in computer hardware over the past 35 years. The 1978 8086 was introduced with a clock speed of 5 MHz and had 29,000 transistors. A six-core Core i7 EE 4960X introduced in 2013 operates at 4 GHz, a speedup of a factor of 800, and has 1.86 billion transistors, about 64,000 times as many as the 8086. Yet the Core i7 EE 4960X is in only a slightly larger package than the 8086 and has a comparable cost.

1.5 EMBEDDED SYSTEMS

The term *embedded system* refers to the use of electronics and software within a product, as opposed to a general-purpose computer, such as a laptop or desktop system. Millions of computers are sold every year, including laptops, personal computers, workstations, servers, mainframes, and supercomputers. In contrast, billions of computer systems are produced each year that are embedded within larger devices. Today, many, perhaps most, devices that use electric power have an embedded computing system. It is likely that in the near future virtually all such devices will have embedded computing systems.

Types of devices with embedded systems are almost too numerous to list. Examples include cell phones, digital cameras, video cameras, calculators, microwave ovens, home security systems, washing machines, lighting systems, thermostats, printers, various automotive systems (e.g., transmission control, cruise control, fuel injection, anti-lock brakes, and suspension systems), tennis rackets, toothbrushes, and numerous types of sensors and actuators in automated systems.

Often, embedded systems are tightly coupled to their environment. This can give rise to real-time constraints imposed by the need to interact with the environment. Constraints, such as required speeds of motion, required precision of measurement, and required time durations, dictate the timing of software operations. If multiple activities must be managed simultaneously, this imposes more complex real-time constraints.

Figure 1.14 shows in general terms an embedded system organization. In addition to the processor and memory, there are a number of elements that differ from the typical desktop or laptop computer:

- There may be a variety of interfaces that enable the system to measure, manipulate, and otherwise interact with the external environment. Embedded systems often interact (sense, manipulate, and communicate) with external world through sensors and actuators and hence are typically reactive systems; a

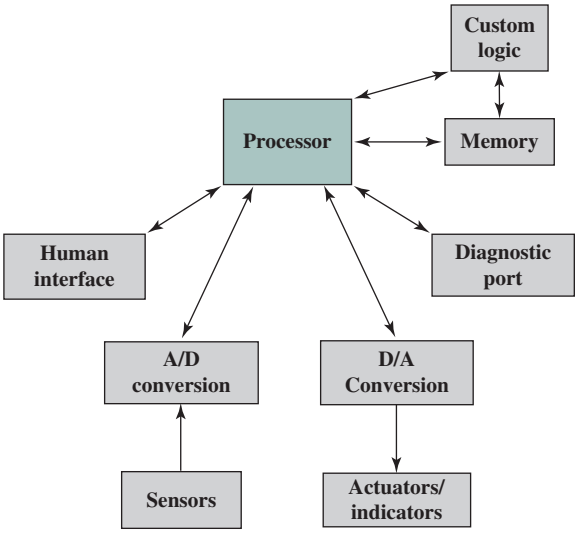


Figure 1.14 Possible Organization of an Embedded System

reactive system is in continual interaction with the environment and executes at a pace determined by that environment.

- The human interface may be as simple as a flashing light or as complicated as real-time robotic vision. In many cases, there is no human interface.
- The diagnostic port may be used for diagnosing the system that is being controlled—not just for diagnosing the computer.
- Special-purpose field programmable (FPGA), application-specific (ASIC), or even nondigital hardware may be used to increase performance or reliability.
- Software often has a fixed function and is specific to the application.
- Efficiency is of paramount importance for embedded systems. They are optimized for energy, code size, execution time, weight and dimensions, and cost.

There are several noteworthy areas of similarity to general-purpose computer systems as well:

- Even with nominally fixed function software, the ability to field upgrade to fix bugs, to improve security, and to add functionality, has become very important for embedded systems, and not just in consumer devices.
- One comparatively recent development has been of embedded system platforms that support a wide variety of apps. Good examples of this are smartphones and audio/visual devices, such as smart TVs.

The Internet of Things

It is worthwhile to separately callout one of the major drivers in the proliferation of embedded systems. The **Internet of things (IoT)** is a term that refers to the expanding

interconnection of smart devices, ranging from appliances to tiny sensors. A dominant theme is the embedding of short-range mobile transceivers into a wide array of gadgets and everyday items, enabling new forms of communication between people and things, and between things themselves. The Internet now supports the interconnection of billions of industrial and personal objects, usually through cloud systems. The objects deliver sensor information, act on their environment, and, in some cases, modify themselves, to create overall management of a larger system, like a factory or city.

The IoT is primarily driven by deeply embedded devices (defined below). These devices are low-bandwidth, low-repetition data-capture, and low-bandwidth data-usage appliances that communicate with each other and provide data via user interfaces. Embedded appliances, such as high-resolution video security cameras, video VoIP phones, and a handful of others, require high-bandwidth streaming capabilities. Yet countless products simply require packets of data to be intermittently delivered.

With reference to the end systems supported, the Internet has gone through roughly four generations of deployment culminating in the IoT:

1. **Information technology (IT):** PCs, servers, routers, firewalls, and so on, bought as IT devices by enterprise IT people and primarily using wired connectivity.
2. **Operational technology (OT):** Machines/appliances with embedded IT built by non-IT companies, such as medical machinery, SCADA (supervisory control and data acquisition), process control, and kiosks, bought as appliances by enterprise OT people and primarily using wired connectivity.
3. **Personal technology:** Smartphones, tablets, and eBook readers bought as IT devices by consumers (employees) exclusively using wireless connectivity and often multiple forms of wireless connectivity.
4. **Sensor/actuator technology:** Single-purpose devices bought by consumers, IT, and OT people exclusively using wireless connectivity, generally of a single form, as part of larger systems.

It is the fourth generation that is usually thought of as the IoT, and it is marked by the use of billions of embedded devices.

Embedded Operating Systems

There are two general approaches to developing an embedded operating system (OS). The first approach is to take an existing OS and adapt it for the embedded application. For example, there are embedded versions of Linux, Windows, and Mac, as well as other commercial and proprietary operating systems specialized for embedded systems. The other approach is to design and implement an OS intended solely for embedded use. An example of the latter is TinyOS, widely used in wireless sensor networks. This topic is explored in depth in [STAL15].

Application Processors versus Dedicated Processors

In this subsection, and the next two, we briefly introduce some terms commonly found in the literature on embedded systems. **Application processors** are defined

by the processor's ability to execute complex operating systems, such as Linux, Android, and Chrome. Thus, the application processor is general-purpose in nature. A good example of the use of an embedded application processor is the smartphone. The embedded system is designed to support numerous apps and perform a wide variety of functions.

Most embedded systems employ a **dedicated processor**, which, as the name implies, is dedicated to one or a small number of specific tasks required by the host device. Because such an embedded system is dedicated to a specific task or tasks, the processor and associated components can be engineered to reduce size and cost.

Microprocessors versus Microcontrollers

As we have seen, early **microprocessor** chips included registers, an ALU, and some sort of control unit or instruction processing logic. As transistor density increased, it became possible to increase the complexity of the instruction set architecture, and ultimately to add memory and more than one processor. Contemporary microprocessor chips, as shown in Figure 1.2, include multiple cores and a substantial amount of cache memory.

A **microcontroller** chip makes a substantially different use of the logic space available. Figure 1.15 shows in general terms the elements typically found on a microcontroller chip. As shown, a microcontroller is a single chip that contains the processor, non-volatile memory for the program (ROM), volatile memory for input and output (RAM), a clock, and an I/O control unit. The processor portion of the microcontroller has a much lower silicon area than other microprocessors and much higher energy efficiency. We examine microcontroller organization in more detail in Section 1.6.

Also called a "computer on a chip," billions of microcontroller units are embedded each year in myriad products from toys to appliances to automobiles. For example, a single vehicle can use 70 or more microcontrollers. Typically, especially for the smaller, less expensive microcontrollers, they are used as dedicated processors for specific tasks. For example, microcontrollers are heavily utilized in automation processes. By providing simple reactions to input, they can control machinery, turn fans on and off, open and close valves, and so forth. They are integral parts of modern industrial technology and are among the most inexpensive ways to produce machinery that can handle extremely complex functionalities.

Microcontrollers come in a range of physical sizes and processing power. Processors range from 4-bit to 32-bit architectures. Microcontrollers tend to be much slower than microprocessors, typically operating in the MHz range rather than the GHz speeds of microprocessors. Another typical feature of a microcontroller is that it does not provide for human interaction. The microcontroller is programmed for a specific task, embedded in its device, and executes as and when required.

Embedded versus Deeply Embedded Systems

We have, in this section, defined the concept of an embedded system. A subset of embedded systems, and a quite numerous subset, is referred to as **deeply embedded systems**. Although this term is widely used in the technical and commercial

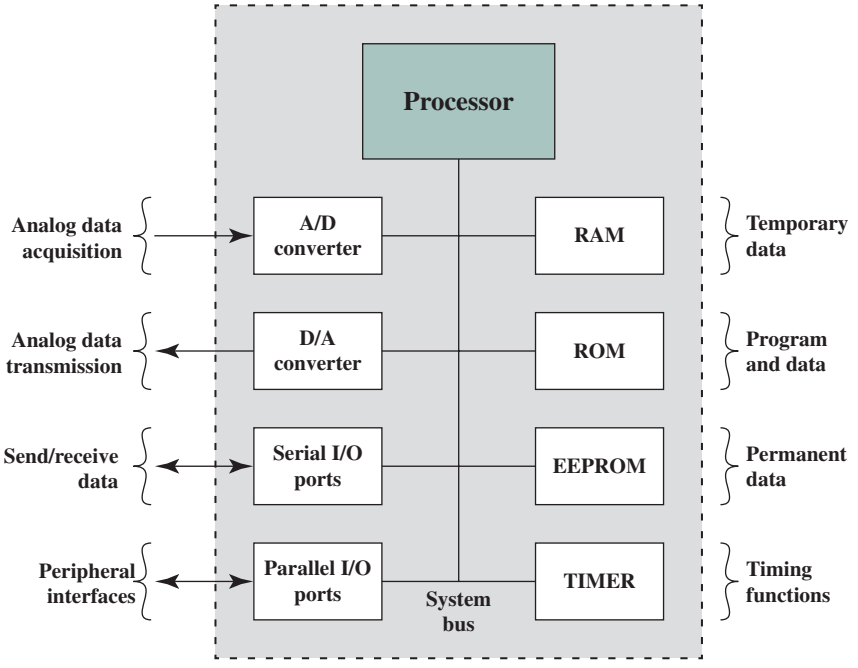


Figure 1.15 Typical Microcontroller Chip Elements

literature, you will search the Internet in vain (or at least I did) for a straightforward definition. Generally, we can say that a deeply embedded system has a processor whose behavior is difficult to observe both by the programmer and the user. A deeply embedded system uses a microcontroller rather than a microprocessor, is not programmable once the program logic for the device has been burned into ROM (read-only memory), and has no interaction with a user.

Deeply embedded systems are dedicated, single-purpose devices that detect something in the environment, perform a basic level of processing, and then do something with the results. Deeply embedded systems often have wireless capability and appear in networked configurations, such as networks of sensors deployed over a large area (e.g., factory, agricultural field). The Internet of things depends heavily on deeply embedded systems. Typically, deeply embedded systems have extreme resource constraints in terms of memory, processor size, time, and power consumption.

1.6 ARM ARCHITECTURE

The ARM architecture refers to a processor architecture that has evolved from RISC design principles and is used in embedded systems. Chapter 15 examines RISC design principles in detail. In this section, we give a brief overview of the ARM architecture.

ARM Evolution

ARM is a family of RISC-based microprocessors and microcontrollers designed by ARM Holdings, Cambridge, England. The company doesn't make processors but instead designs microprocessor and multicore architectures and licenses them to manufacturers. Specifically, ARM Holdings has two types of licensable products: processors and processor architectures. For processors, the customer buys the rights to use ARM-supplied design in their own chips. For a processor architecture, the customer buys the rights to design their own processor compliant with ARM's architecture.

ARM chips are high-speed processors that are known for their small die size and low power requirements. They are widely used in smartphones and other handheld devices, including game systems, as well as a large variety of consumer products. ARM chips are the processors in Apple's popular iPod and iPhone devices, and are used in virtually all Android smartphones as well. ARM is probably the most widely used embedded processor architecture and indeed the most widely used processor architecture of any kind in the world [VANC14].

The origins of ARM technology can be traced back to the British-based Acorn Computers company. In the early 1980s, Acorn was awarded a contract by the British Broadcasting Corporation (BBC) to develop a new microcomputer architecture for the BBC Computer Literacy Project. The success of this contract enabled Acorn to go on to develop the first commercial RISC processor, the Acorn RISC Machine (ARM). The first version, ARM1, became operational in 1985 and was used for internal research and development as well as being used as a coprocessor in the BBC machine.

In this early stage, Acorn used the company VLSI Technology to do the actual fabrication of the processor chips. VLSI was licensed to market the chip on its own and had some success in getting other companies to use the ARM in their products, particularly as an embedded processor.

The ARM design matched a growing commercial need for a high-performance, low-power-consumption, small-size, and low-cost processor for embedded applications. But further development was beyond the scope of Acorn's capabilities. Accordingly, a new company was organized, with Acorn, VLSI, and Apple Computer as founding partners, known as ARM Ltd. The Acorn RISC Machine became Advanced RISC Machines.[12]

Instruction Set Architecture

The ARM instruction set is highly regular, designed for efficient implementation of the processor and efficient execution. All instructions are 32 bits long and follow a regular format. This makes the ARM ISA suitable for implementation over a wide range of products.

Augmenting the basic ARM ISA is the Thumb instruction set, which is a re-encoded subset of the ARM instruction set. Thumb is designed to increase the performance of ARM implementations that use a 16-bit or narrower memory data bus,

[12]The company dropped the designation *Advanced RISC Machines* in the late 1990s. It is now simply known as the ARM architecture.

and to allow better code density than provided by the ARM instruction set. The Thumb instruction set contains a subset of the ARM 32-bit instruction set recoded into 16-bit instructions. The current defined version is Thumb-2.

The ARM and Thumb-2 ISAs are discussed in Chapters 12 and 13.

ARM Products

ARM Holdings licenses a number of specialized microprocessors and related technologies, but the bulk of their product line is the Cortex family of microprocessor architectures. There are three Cortex architectures, conveniently labeled with the initials A, R, and M.

CORTEX-A/CORTEX-A50 The Cortex-A and Cortex-A50 are application processors, intended for mobile devices such as smartphones and eBook readers, as well as consumer devices such as digital TV and home gateways (e.g., DSL and cable Internet modems). These processors run at higher clock frequency (over 1 GHz), and support a memory management unit (MMU), which is required for full feature OSs such as Linux, Android, MS Windows, and mobile OSs. An MMU is a hardware module that supports virtual memory and paging by translating virtual addresses into physical addresses; this topic is explored in Chapter 8.

The two architectures use both the ARM and Thumb-2 instruction sets; the principal difference is that the Cortex-A is a 32-bit machine, and the Cortex-A50 is a 64-bit machine.

CORTEX-R The Cortex-R is designed to support real-time applications, in which the timing of events needs to be controlled with rapid response to events. They can run at a fairly high clock frequency (e.g., 200MHz to 800MHz) and have very low response latency. The Cortex-R includes enhancements both to the instruction set and to the processor organization to support deeply embedded real-time devices. Most of these processors do not have MMU; the limited data requirements and the limited number of simultaneous processes eliminates the need for elaborate hardware and software support for virtual memory. The Cortex-R does have a Memory Protection Unit (MPU), cache, and other memory features designed for industrial applications. An MPU is a hardware module that prohibits one program in memory from accidentally accessing memory assigned to another active program. Using various methods, a protective boundary is created around the program, and instructions within the program are prohibited from referencing data outside of that boundary.

Examples of embedded systems that would use the Cortex-R are automotive braking systems, mass storage controllers, and networking and printing devices.

CORTEX-M Cortex-M series processors have been developed primarily for the microcontroller domain where the need for fast, highly deterministic interrupt management is coupled with the desire for extremely low gate count and lowest possible power consumption. As with the Cortex-R series, the Cortex-M architecture has an MPU but no MMU. The Cortex-M uses only the Thumb-2 instruction set. The market for the Cortex-M includes IoT devices, wireless sensor/actuator networks used in factories and other enterprises, automotive body electronics, and so on.

There are currently four versions of the Cortex-M series:

- **Cortex-M0:** Designed for 8- and 16-bit applications, this model emphasizes low cost, ultra low power, and simplicity. It is optimized for small silicon die size (starting from 12k gates) and use in the lowest cost chips.
- **Cortex-M0+:** An enhanced version of the M0 that is more energy efficient.
- **Cortex-M3:** Designed for 16- and 32-bit applications, this model emphasizes performance and energy efficiency. It also has comprehensive debug and trace features to enable software developers to develop their applications quickly.
- **Cortex-M4:** This model provides all the features of the Cortex-M3, with additional instructions to support digital signal processing tasks.

In this text, we will primarily use the ARM Cortex-M3 as our example embedded system processor. It is the best suited of all ARM models for general-purpose microcontroller use. The Cortex-M3 is used by a variety of manufacturers of microcontroller products. Initial microcontroller devices from lead partners already combine the Cortex-M3 processor with flash, SRAM, and multiple peripherals to provide a competitive offering at the price of just $1.

Figure 1.16 provides a block diagram of the EFM32 microcontroller from Silicon Labs. The figure also shows detail of the Cortex-M3 processor and core components. We examine each level in turn.

The **Cortex-M3 core** makes use of separate buses for instructions and data. This arrangement is sometimes referred to as a Harvard architecture, in contrast with the von Neumann architecture, which uses the same signal buses and memory for both instructions and data. By being able to read both an instruction and data from memory at the same time, the Cortex-M3 processor can perform many operations in parallel, speeding application execution. The core contains a decoder for Thumb instructions, an advanced ALU with support for hardware multiply and divide, control logic, and interfaces to the other components of the processor. In particular, there is an interface to the nested vector interrupt controller (NVIC) and the embedded trace macrocell (ETM) module.

The core is part of a module called the **Cortex-M3 processor**. This term is somewhat misleading, because typically in the literature, the terms core and processor are viewed as equivalent. In addition to the core, the processor includes the following elements:

- **NVIC:** Provides configurable interrupt handling abilities to the processor. It facilitates low-latency exception and interrupt handling, and controls power management.
- **ETM:** An optional debug component that enables reconstruction of program execution. The ETM is designed to be a high-speed, low-power debug tool that only supports instruction trace.
- **Debug access port (DAP):** This provides an interface for external debug access to the processor.
- **Debug logic:** Basic debug functionality includes processor halt, single-step, processor core register access, unlimited software breakpoints, and full system memory access.

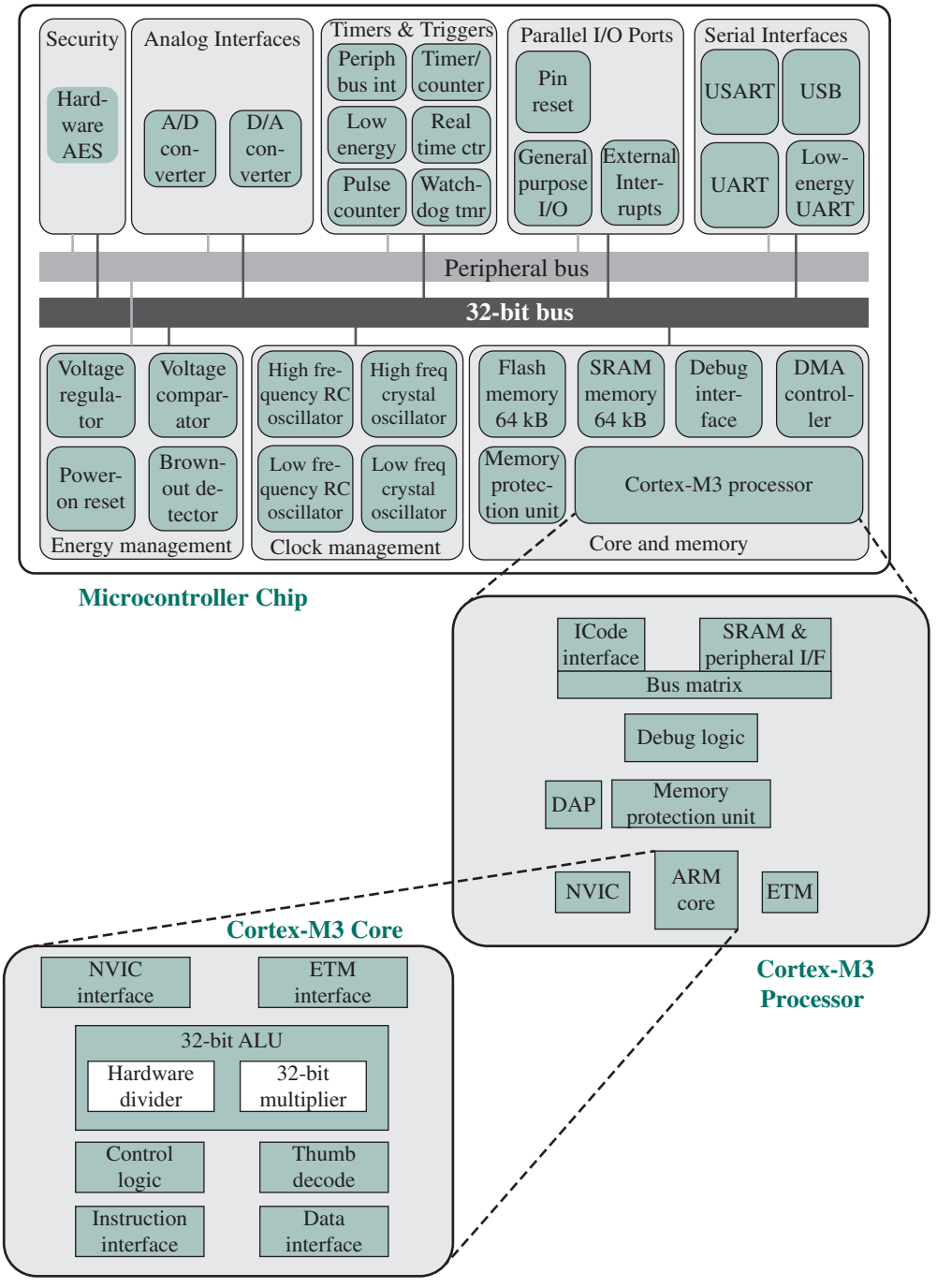


Figure 1.16 Typical Microcontroller Chip Based on Cortex-M3

- **ICode interface:** Fetches instructions from the code memory space.
- **SRAM & peripheral interface:** Read/write interface to data memory and peripheral devices.
- **Bus matrix:** Connects the core and debug interfaces to external buses on the microcontroller.
- **Memory protection unit:** Protects critical data used by the operating system from user applications, separating processing tasks by disallowing access to each other's data, disabling access to memory regions, allowing memory regions to be defined as read-only, and detecting unexpected memory accesses that could potentially break the system.

The upper part of Figure 1.16 shows the block diagram of a typical microcontroller built with the Cortex-M3, in this case the EFM32 microcontroller. This microcontroller is marketed for use in a wide variety of devices, including energy, gas, and water metering; alarm and security systems; industrial automation devices; home automation devices; smart accessories; and health and fitness devices. The silicon chip consists of 10 main areas:[13]

- **Core and memory:** This region includes the Cortex-M3 processor, static RAM (SRAM) data memory,[14] and flash memory[15] for storing program instructions and nonvarying application data. Flash memory is nonvolatile (data is not lost when power is shut off) and so is ideal for this purpose. The SRAM stores variable data. This area also includes a debug interface, which makes it easy to reprogram and update the system in the field.
- **Parallel I/O ports:** Configurable for a variety of parallel I/O schemes.
- **Serial interfaces:** Supports various serial I/O schemes.
- **Analog interfaces:** Analog-to-digital and digital-to-analog logic to support sensors and actuators.
- **Timers and triggers:** Keeps track of timing and counts events, generates output waveforms, and triggers timed actions in other peripherals.
- **Clock management:** Controls the clocks and oscillators on the chip. Multiple clocks and oscillators are used to minimize power consumption and provide short startup times.
- **Energy management:** Manages the various low-energy modes of operation of the processor and peripherals to provide real-time management of the energy needs so as to minimize energy consumption.
- **Security:** The chip includes a hardware implementation of the Advanced Encryption Standard (AES).

---

[13]This discussion does not go into details about all of the individual modules; for the interested reader, an in-depth discussion is provided in the document EFM32G200.pdf, available at box.com/COA10e.

[14]Static RAM (SRAM) is a form of random-access memory used for cache memory; see Chapter 5.

[15]Flash memory is a versatile form of memory used both in microcontrollers and as external memory; it is discussed in Chapter 6.

- **32-bit bus:** Connects all of the components on the chip.
- **Peripheral bus:** A network which lets the different peripheral module communicate directly with each other without involving the processor. This supports timing-critical operation and reduces software overhead.

Comparing Figure 1.16 with Figure 1.2, you will see many similarities and the same general hierarchical structure. Note, however, that the top level of a microcontroller computer system is a single chip, whereas for a multicore computer, the top level is a motherboard containing a number of chips. Another noteworthy difference is that there is no cache, neither in the Cortex-M3 processor nor in the microcontroller as a whole, which plays an important role if the code or data resides in external memory. Though the number of cycles to read the instruction or data varies depending on cache hit or miss, the cache greatly improves the performance when external memory is used. Such overhead is not needed for a microcontroller.

1.7 CLOUD COMPUTING

Although the general concepts for cloud computing go back to the 1950s, cloud computing services first became available in the early 2000s, particularly targeted at large enterprises. Since then, cloud computing has spread to small and medium size businesses, and most recently to consumers. Apple's iCloud was launched in 2012 and had 20 million users within a week of launch. Evernote, the cloud-based notetaking and archiving service, launched in 2008, approached 100 million users in less than 6 years. In this section, we provide a brief overview. Cloud computing is examined in more detail in Chapter 17.

Basic Concepts

There is an increasingly prominent trend in many organizations to move a substantial portion or even all information technology (IT) operations to an Internet-connected infrastructure known as enterprise cloud computing. At the same time, individual users of PCs and mobile devices are relying more and more on cloud computing services to backup data, synch devices, and share, using personal cloud computing. NIST defines cloud computing, in NIST SP-800-145 (*The NIST Definition of Cloud Computing*), as follows:

> **Cloud computing:** A model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Basically, with cloud computing, you get economies of scale, professional network management, and professional security management. These features can be attractive to companies large and small, government agencies, and individual PC and mobile users. The individual or company only needs to pay for the storage

capacity and services they need. The user, be it company or individual, doesn't have the hassle of setting up a database system, acquiring the hardware they need, doing maintenance, and backing up the data—all these are part of the cloud service.

In theory, another big advantage of using cloud computing to store your data and share it with others is that the cloud provider takes care of security. Alas, the customer is not always protected. There have been a number of security failures among cloud providers. Evernote made headlines in early 2013 when it told all of its users to reset their passwords after an intrusion was discovered.

Cloud networking refers to the networks and network management functionality that must be in place to enable cloud computing. Most cloud computing solutions rely on the Internet, but that is only a piece of the networking infrastructure. One example of cloud networking is the provisioning of high-performance and/or high-reliability networking between the provider and subscriber. In this case, some or all of the traffic between an enterprise and the cloud bypasses the Internet and uses dedicated private network facilities owned or leased by the cloud service provider. More generally, cloud networking refers to the collection of network capabilities required to access a cloud, including making use of specialized services over the Internet, linking enterprise data centers to a cloud, and using firewalls and other network security devices at critical points to enforce access security policies.

We can think of **cloud storage** as a subset of cloud computing. In essence, cloud storage consists of database storage and database applications hosted remotely on cloud servers. Cloud storage enables small businesses and individual users to take advantage of data storage that scales with their needs and to take advantage of a variety of database applications without having to buy, maintain, and manage the storage assets.

Cloud Services

The essential purpose of cloud computing is to provide for the convenient rental of computing resources. A cloud service provider (CSP) maintains computing and data storage resources that are available over the Internet or private networks. Customers can rent a portion of these resources as needed. Virtually all cloud service is provided using one of three models (Figure 1.17): SaaS, PaaS, and IaaS, which we examine in this section.

SOFTWARE AS A SERVICE (SAAS) As the name implies, a SaaS cloud provides service to customers in the form of software, specifically application software, running on and accessible in the cloud. SaaS follows the familiar model of Web services, in this case applied to cloud resources. SaaS enables the customer to use the cloud provider's applications running on the provider's cloud infrastructure. The applications are accessible from various client devices through a simple interface such as a Web browser. Instead of obtaining desktop and server licenses for software products it uses, an enterprise obtains the same functions from the cloud service. SaaS saves the complexity of software installation, maintenance, upgrades, and patches. Examples of services at this level are Gmail, Google's e-mail service, and Salesforce.com, which help firms keep track of their customers.

Common subscribers to SaaS are organizations that want to provide their employees with access to typical office productivity software, such as document

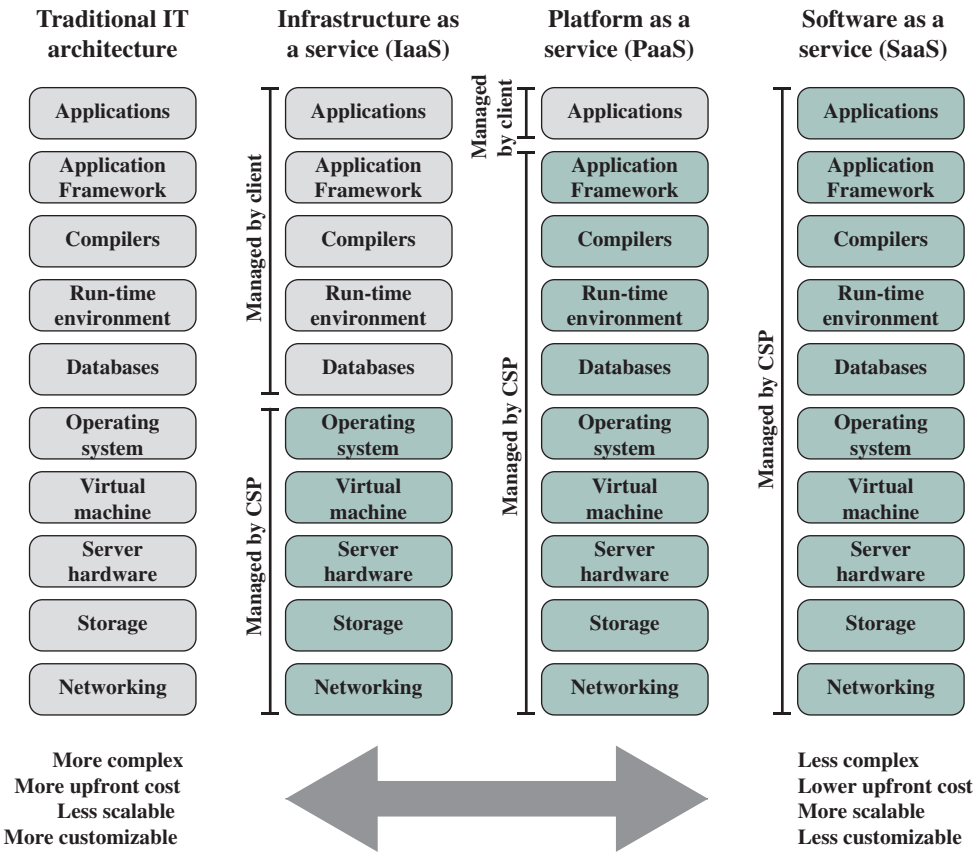


Figure 1.17 Alternative Information Technology Architectures

management and email. Individuals also commonly use the SaaS model to acquire cloud resources. Typically, subscribers use specific applications on demand. The cloud provider also usually offers data-related features such as automatic backup and data sharing between subscribers.

PLATFORM AS A SERVICE (PAAS) A PaaS cloud provides service to customers in the form of a platform on which the customer's applications can run. PaaS enables the customer to deploy onto the cloud infrastructure containing customer-created or acquired applications. A PaaS cloud provides useful software building blocks, plus a number of development tools, such as programming languages, run-time environments, and other tools that assist in deploying new applications. In effect, PaaS is an operating system in the cloud. PaaS is useful for an organization that wants to develop new or tailored applications while paying for the needed computing resources only as needed and only for as long as needed. Google App Engine and the Salesforce1 Platform from Salesforce.com are examples of PaaS.

INFRASTRUCTURE AS A SERVICE (IAAS) With IaaS, the customer has access to the underlying cloud infrastructure. IaaS provides virtual machines and other abstracted hardware and operating systems, which may be controlled through a service application programming interface (API). IaaS offers the customer processing, storage, networks, and other fundamental computing resources so that the customer is able to deploy and run arbitrary software, which can include operating systems and applications. IaaS enables customers to combine basic computing services, such as number crunching and data storage, to build highly adaptable computer systems. Examples of IaaS are Amazon Elastic Compute Cloud (Amazon EC2) and Windows Azure.

1.8 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

application processor	gate	microprocessor
arithmetic and logic unit (ALU)	infrastructure as a service (IaaS)	motherboard
ARM	input–output (I/O)	multicore
central processing unit (CPU)	instruction set architecture (ISA)	multicore processor
chip	integrated circuit	original equipment manufacturer (OEM)
cloud computing	Intel x86	platform as a service (PaaS)
cloud networking	Internet of things (IoT)	printed circuit board
cloud storage	main memory	processor
computer architecture	memory cell	registers
computer organization	memory management unit (MMU)	semiconductor
control unit	memory protection unit (MPU)	semiconductor memory
core	microcontroller	software as a service (SaaS)
dedicated processor	microelectronics	system bus
deeply embedded system		system interconnection
embedded system		vacuum tubes

Review Questions

1.1 What, in general terms, is the distinction between computer organization and computer architecture?

1.2 What, in general terms, is the distinction between computer structure and computer function?

1.3 What are the four main functions of a computer?

1.4 List and briefly define the main structural components of a computer.

1.5 List and briefly define the main structural components of a processor.

1.6 What is a stored program computer?

1.7 Explain Moore's law.

1.8 List and explain the key characteristics of a computer family.

1.9 What is the key distinguishing feature of a microprocessor?

Problems

1.1 You are to write an IAS program to compute the results of the following equation.

$$Y = \sum_{X=1}^{N} X$$

Assume that the computation does not result in an arithmetic overflow and that X, Y, and N are positive integers with $N \geq 1$. *Note*: The IAS did not have assembly language, only machine language.

a. Use the equation $\text{Sum}(Y) = \frac{N(N+1)}{2}$ when writing the IAS program.

b. Do it the "hard way," without using the equation from part (a).

1.2 **a.** On the IAS, what would the machine code instruction look like to load the contents of memory address 2 to the accumulator?

b. How many trips to memory does the CPU need to make to complete this instruction during the instruction cycle?

1.3 On the IAS, describe in English the process that the CPU must undertake to read a value from memory and to write a value to memory in terms of what is put into the MAR, MBR, address bus, data bus, and control bus.

1.4 Given the memory contents of the IAS computer shown below,

Address	Contents
08A	010FA210FB
08B	010FA0F08D
08C	020FA210FB

show the assembly language code for the program, starting at address 08A. Explain what this program does.

1.5 In Figure 1.6, indicate the width, in bits, of each data path (e.g., between AC and ALU).

1.6 In the IBM 360 Models 65 and 75, addresses are staggered in two separate main memory units (e.g., all even-numbered words in one unit and all odd-numbered words in another). What might be the purpose of this technique?

1.7 The relative performance of the IBM 360 Model 75 is 50 times that of the 360 Model 30, yet the instruction cycle time is only 5 times as fast. How do you account for this discrepancy?

1.8 While browsing at Billy Bob's computer store, you overhear a customer asking Billy Bob what is the fastest computer in the store that he can buy. Billy Bob replies, "You're looking at our Macintoshes. The fastest Mac we have runs at a clock speed of 1.2 GHz. If you really want the fastest machine, you should buy our 2.4-GHz Intel Pentium IV instead." Is Billy Bob correct? What would you say to help this customer?

1.9 The ENIAC, a precursor to the ISA machine, was a decimal machine, in which each register was represented by a ring of 10 vacuum tubes. At any time, only one vacuum tube was in the ON state, representing one of the 10 decimal digits. Assuming that ENIAC had the capability to have multiple vacuum tubes in the ON and OFF state simultaneously, why is this representation "wasteful" and what range of integer values could we represent using the 10 vacuum tubes?

1.10 For each of the following examples, determine whether this is an embedded system, explaining why or why not.

a. Are programs that understand physics and/or hardware embedded? For example, one that uses finite-element methods to predict fluid flow over airplane wings?

b. Is the internal microprocessor controlling a disk drive an example of an embedded system?

c. I/O drivers control hardware, so does the presence of an I/O driver imply that the computer executing the driver is embedded?
d. Is a PDA (Personal Digital Assistant) an embedded system?
e. Is the microprocessor controlling a cell phone an embedded system?
f. Are the computers in a big phased-array radar considered embedded? These radars are 10-story buildings with one to three 100-foot diameter radiating patches on the sloped sides of the building.
g. Is a traditional flight management system (FMS) built into an airplane cockpit considered embedded?
h. Are the computers in a hardware-in-the-loop (HIL) simulator embedded?
i. Is the computer controlling a pacemaker in a person's chest an embedded computer?
j. Is the computer controlling fuel injection in an automobile engine embedded?

CHAPTER 2

PERFORMANCE ISSUES

2.1 Designing for Performance
- Microprocessor Speed
- Performance Balance
- Improvements in Chip Organization and Architecture

2.2 Multicore, MICs, and GPGPUs

2.3 Two Laws that Provide Insight: Amdahl's Law and Little's Law
- Amdahl's Law
- Little's Law

2.4 Basic Measures of Computer Performance
- Clock Speed
- Instruction Execution Rate

2.5 Calculating the Mean
- Arithmetic Mean
- Harmonic Mean
- Geometric Mean

2.6 Benchmarks and SPEC
- Benchmark Principles
- SPEC Benchmarks

2.7 Key Terms, Review Questions, and Problems

Learning Objectives

After studying this chapter, you should be able to:

- Understand the key performance issues that relate to computer design.
- Explain the reasons for the move to multicore organization, and understand the trade-off between cache and processor resources on a single chip.
- Distinguish among multicore, MIC, and GPGPU organizations.
- Summarize some of the issues in computer performance assessment.
- Discuss the SPEC benchmarks.
- Explain the differences among arithmetic, harmonic, and geometric means.

This chapter addresses the issue of computer system performance. We begin with a consideration of the need for balanced utilization of computer resources, which provides a perspective that is useful throughout the book. Next we look at contemporary computer organization designs intended to provide performance to meet current and projected demand. Finally, we look at tools and models that have been developed to provide a means of assessing comparative computer system performance.

2.1 DESIGNING FOR PERFORMANCE

Year by year, the cost of computer systems continues to drop dramatically, while the performance and capacity of those systems continue to rise equally dramatically. Today's laptops have the computing power of an IBM mainframe from 10 or 15 years ago. Thus, we have virtually "free" computer power. Processors are so inexpensive that we now have microprocessors we throw away. The digital pregnancy test is an example (used once and then thrown away). And this continuing technological revolution has enabled the development of applications of astounding complexity and power. For example, desktop applications that require the great power of today's microprocessor-based systems include

- Image processing
- Three-dimensional rendering
- Speech recognition
- Videoconferencing
- Multimedia authoring
- Voice and video annotation of files
- Simulation modeling

Workstation systems now support highly sophisticated engineering and scientific applications and have the capacity to support image and video applications. In addition, businesses are relying on increasingly powerful servers to handle transaction and database processing and to support massive client/server networks that have replaced the huge mainframe computer centers of yesteryear. As well, cloud service

providers use massive high-performance banks of servers to satisfy high-volume, high-transaction-rate applications for a broad spectrum of clients.

What is fascinating about all this from the perspective of computer organization and architecture is that, on the one hand, the basic building blocks for today's computer miracles are virtually the same as those of the IAS computer from over 50 years ago, while on the other hand, the techniques for squeezing the maximum performance out of the materials at hand have become increasingly sophisticated.

This observation serves as a guiding principle for the presentation in this book. As we progress through the various elements and components of a computer, two objectives are pursued. First, the book explains the fundamental functionality in each area under consideration, and second, the book explores those techniques required to achieve maximum performance. In the remainder of this section, we highlight some of the driving factors behind the need to design for performance.

Microprocessor Speed

What gives Intel x86 processors or IBM mainframe computers such mind-boggling power is the relentless pursuit of speed by processor chip manufacturers. The evolution of these machines continues to bear out Moore's law, described in Chapter 1. So long as this law holds, chipmakers can unleash a new generation of chips every three years—with four times as many transistors. In memory chips, this has quadrupled the capacity of **dynamic random-access memory (DRAM)**, still the basic technology for computer main memory, every three years. In microprocessors, the addition of new circuits, and the speed boost that comes from reducing the distances between them, has improved performance four- or fivefold every three years or so since Intel launched its x86 family in 1978.

But the raw speed of the microprocessor will not achieve its potential unless it is fed a constant stream of work to do in the form of computer instructions. Anything that gets in the way of that smooth flow undermines the power of the processor. Accordingly, while the chipmakers have been busy learning how to fabricate chips of greater and greater density, the processor designers must come up with ever more elaborate techniques for feeding the monster. Among the techniques built into contemporary processors are the following:

- **Pipelining:** The execution of an instruction involves multiple stages of operation, including fetching the instruction, decoding the opcode, fetching operands, performing a calculation, and so on. Pipelining enables a processor to work simultaneously on multiple instructions by performing a different phase for each of the multiple instructions at the same time. The processor overlaps operations by moving data or instructions into a conceptual pipe with all stages of the pipe processing simultaneously. For example, while one instruction is being executed, the computer is decoding the next instruction. This is the same principle as seen in an assembly line.
- **Branch prediction:** The processor looks ahead in the instruction code fetched from memory and predicts which branches, or groups of instructions, are likely to be processed next. If the processor guesses right most of the time, it can prefetch the correct instructions and buffer them so that the processor is kept busy. The more sophisticated examples of this strategy predict not just

the next branch but multiple branches ahead. Thus, branch prediction potentially increases the amount of work available for the processor to execute.

- **Superscalar execution:** This is the ability to issue more than one instruction in every processor clock cycle. In effect, multiple parallel pipelines are used.
- **Data flow analysis:** The processor analyzes which instructions are dependent on each other's results, or data, to create an optimized schedule of instructions. In fact, instructions are scheduled to be executed when ready, independent of the original program order. This prevents unnecessary delay.
- **Speculative execution:** Using branch prediction and data flow analysis, some processors speculatively execute instructions ahead of their actual appearance in the program execution, holding the results in temporary locations. This enables the processor to keep its execution engines as busy as possible by executing instructions that are likely to be needed.

These and other sophisticated techniques are made necessary by the sheer power of the processor. Collectively they make it possible to execute many instructions per processor cycle, rather than to take many cycles per instruction.

Performance Balance

While processor power has raced ahead at breakneck speed, other critical components of the computer have not kept up. The result is a need to look for performance balance: an adjustment/tuning of the organization and architecture to compensate for the mismatch among the capabilities of the various components.

The problem created by such mismatches is particularly critical at the interface between processor and main memory. While processor speed has grown rapidly, the speed with which data can be transferred between main memory and the processor has lagged badly. The interface between processor and main memory is the most crucial pathway in the entire computer because it is responsible for carrying a constant flow of program instructions and data between memory chips and the processor. If memory or the pathway fails to keep pace with the processor's insistent demands, the processor stalls in a wait state, and valuable processing time is lost.

A system architect can attack this problem in a number of ways, all of which are reflected in contemporary computer designs. Consider the following examples:

- Increase the number of bits that are retrieved at one time by making DRAMs "wider" rather than "deeper" and by using wide bus data paths.
- Change the DRAM interface to make it more efficient by including a cache[1] or other buffering scheme on the DRAM chip.
- Reduce the frequency of memory access by incorporating increasingly complex and efficient cache structures between the processor and main memory. This includes the incorporation of one or more caches on the processor chip as well as on an off-chip cache close to the processor chip.

[1]A cache is a relatively small fast memory interposed between a larger, slower memory and the logic that accesses the larger memory. The cache holds recently accessed data and is designed to speed up subsequent access to the same data. Caches are discussed in Chapter 4.

- Increase the interconnect bandwidth between processors and memory by using higher-speed buses and a hierarchy of buses to buffer and structure data flow.

Another area of design focus is the handling of I/O devices. As computers become faster and more capable, more sophisticated applications are developed that support the use of peripherals with intensive I/O demands. Figure 2.1 gives some examples of typical peripheral devices in use on personal computers and workstations. These devices create tremendous data throughput demands. While the current generation of processors can handle the data pumped out by these devices, there remains the problem of getting that data moved between processor and peripheral. Strategies here include caching and buffering schemes plus the use of higher-speed interconnection buses and more elaborate interconnection structures. In addition, the use of multiple-processor configurations can aid in satisfying I/O demands.

The key in all this is balance. Designers constantly strive to balance the throughput and processing demands of the processor components, main memory, I/O devices, and the interconnection structures. This design must constantly be rethought to cope with two constantly evolving factors:

- The rate at which performance is changing in the various technology areas (processor, buses, memory, peripherals) differs greatly from one type of element to another.
- New applications and new peripheral devices constantly change the nature of the demand on the system in terms of typical instruction profile and the data access patterns.

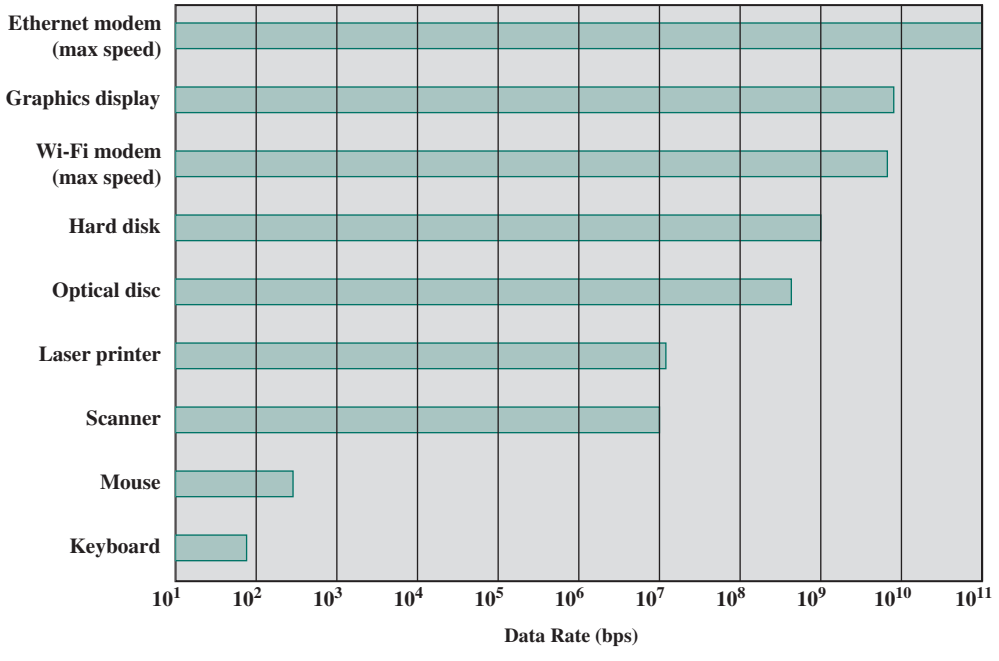


Figure 2.1 Typical I/O Device Data Rates

Thus, computer design is a constantly evolving art form. This book attempts to present the fundamentals on which this art form is based and to present a survey of the current state of that art.

Improvements in Chip Organization and Architecture

As designers wrestle with the challenge of balancing processor performance with that of main memory and other computer components, the need to increase processor speed remains. There are three approaches to achieving increased processor speed:

- Increase the hardware speed of the processor. This increase is fundamentally due to shrinking the size of the logic gates on the processor chip, so that more gates can be packed together more tightly and to increasing the clock rate. With gates closer together, the propagation time for signals is significantly reduced, enabling a speeding up of the processor. An increase in clock rate means that individual operations are executed more rapidly.
- Increase the size and speed of caches that are interposed between the processor and main memory. In particular, by dedicating a portion of the processor chip itself to the cache, cache access times drop significantly.
- Make changes to the processor organization and architecture that increase the effective speed of instruction execution. Typically, this involves using parallelism in one form or another.

Traditionally, the dominant factor in performance gains has been in increases in clock speed due and logic density. However, as clock speed and logic density increase, a number of obstacles become more significant [INTE04]:

- **Power:** As the density of logic and the clock speed on a chip increase, so does the power density (Watts/cm^2). The difficulty of dissipating the heat generated on high-density, high-speed chips is becoming a serious design issue [GIBB04, BORK03].
- **RC delay:** The speed at which electrons can flow on a chip between transistors is limited by the resistance and capacitance of the metal wires connecting them; specifically, delay increases as the RC product increases. As components on the chip decrease in size, the wire interconnects become thinner, increasing resistance. Also, the wires are closer together, increasing capacitance.
- **Memory latency and throughput:** Memory access speed (latency) and transfer speed (throughput) lag processor speeds, as previously discussed.

Thus, there will be more emphasis on organization and architectural approaches to improving performance. These techniques are discussed in later chapters of the text.

Beginning in the late 1980s, and continuing for about 15 years, two main strategies have been used to increase performance beyond what can be achieved simply by increasing clock speed. First, there has been an increase in cache capacity. There are now typically two or three levels of cache between the processor and main memory. As chip density has increased, more of the cache memory has been incorporated on the chip, enabling faster cache access. For example, the original Pentium

chip devoted about 10% of on-chip area to a cache. Contemporary chips devote over half of the chip area to caches. And, typically, about three-quarters of the other half is for pipeline-related control and buffering.

Second, the instruction execution logic within a processor has become increasingly complex to enable parallel execution of instructions within the processor. Two noteworthy design approaches have been pipelining and superscalar. A pipeline works much as an assembly line in a manufacturing plant enabling different stages of execution of different instructions to occur at the same time along the pipeline. A superscalar approach in essence allows multiple pipelines within a single processor, so that instructions that do not depend on one another can be executed in parallel.

By the mid to late 90s, both of these approaches were reaching a point of diminishing returns. The internal organization of contemporary processors is exceedingly complex and is able to squeeze a great deal of parallelism out of the instruction stream. It seems likely that further significant increases in this direction will be relatively modest [GIBB04]. With three levels of cache on the processor chip, each level providing substantial capacity, it also seems that the benefits from the cache are reaching a limit.

However, simply relying on increasing clock rate for increased performance runs into the power dissipation problem already referred to. The faster the clock rate, the greater the amount of power to be dissipated, and some fundamental physical limits are being reached.

Figure 2.2 illustrates the concepts we have been discussing.[2] The top line shows that, as per Moore's Law, the number of transistors on a single chip continues to

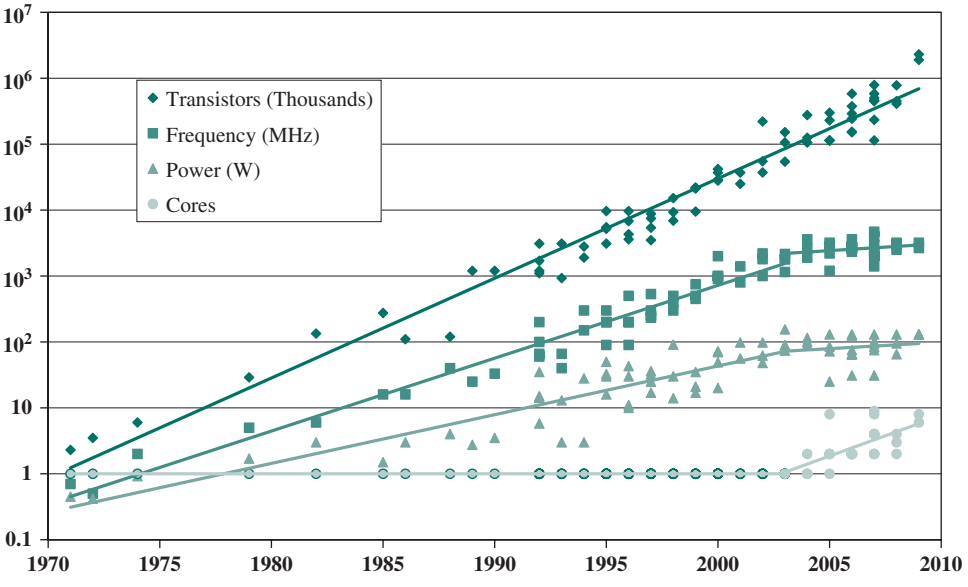


Figure 2.2 Processor Trends

[2]I am grateful to Professor Kathy Yelick of UC Berkeley, who provided this graph.

grow exponentially.[3] Meanwhile, the clock speed has leveled off, in order to prevent a further rise in power. To continue increasing performance, designers have had to find ways of exploiting the growing number of transistors other than simply building a more complex processor. The response in recent years has been the development of the multicore computer chip.

2.2 MULTICORE, MICS, AND GPGPUS

With all of the difficulties cited in the preceding section in mind, designers have turned to a fundamentally new approach to improving performance: placing multiple processors on the same chip, with a large shared cache. The use of multiple processors on the same chip, also referred to as multiple cores, or **multicore**, provides the potential to increase performance without increasing the clock rate. Studies indicate that, within a processor, the increase in performance is roughly proportional to the square root of the increase in complexity [BORK03]. But if the software can support the effective use of multiple processors, then doubling the number of processors almost doubles performance. Thus, the strategy is to use two simpler processors on the chip rather than one more complex processor.

In addition, with two processors, larger caches are justified. This is important because the power consumption of memory logic on a chip is much less than that of processing logic.

As the logic density on chips continues to rise, the trend for both more cores and more cache on a single chip continues. Two-core chips were quickly followed by four-core chips, then 8, then 16, and so on. As the caches became larger, it made performance sense to create two and then three levels of cache on a chip, with initially, the first-level cache dedicated to an individual processor and levels two and three being shared by all the processors. It is now common for the second-level cache to also be private to each core.

Chip manufacturers are now in the process of making a huge leap forward in the number of cores per chip, with more than 50 cores per chip. The leap in performance as well as the challenges in developing software to exploit such a large number of cores has led to the introduction of a new term: **many integrated core (MIC)**.

The multicore and MIC strategy involves a homogeneous collection of general-purpose processors on a single chip. At the same time, chip manufacturers are pursuing another design option: a chip with multiple general-purpose processors plus **graphics processing units (GPUs)** and specialized cores for video processing and other tasks. In broad terms, a GPU is a core designed to perform parallel operations on graphics data. Traditionally found on a plug-in graphics card (display adapter), it is used to encode and render 2D and 3D graphics as well as process video.

Since GPUs perform parallel operations on multiple sets of data, they are increasingly being used as vector processors for a variety of applications that require repetitive computations. This blurs the line between the GPU and the CPU

[3]The observant reader will note that the transistor count values in this figure are significantly less than those of Figure 1.12. That latter figure shows the transistor count for a form of main memory known as DRAM (discussed in Chapter 5), which supports higher transistor density than processor chips.

[AROR12, FATA08, PROP11]. When a broad range of applications are supported by such a processor, the term **general-purpose computing on GPUs (GPGPU)** is used.

We explore design characteristics of multicore computers in Chapter 18 and GPGPUs in Chapter 19.

2.3 TWO LAWS THAT PROVIDE INSIGHT: AHMDAHL'S LAW AND LITTLE'S LAW

In this section, we look at two equations, called "laws." The two laws are unrelated but both provide insight into the performance of parallel systems and multicore systems.

Amdahl's Law

Computer system designers look for ways to improve system performance by advances in technology or change in design. Examples include the use of parallel processors, the use of a memory cache hierarchy, and speedup in memory access time and I/O transfer rate due to technology improvements. In all of these cases, it is important to note that a speedup in one aspect of the technology or design does not result in a corresponding improvement in performance. This limitation is succinctly expressed by Amdahl's law.

Amdahl's law was first proposed by Gene Amdahl in 1967 ([AMDA67], [AMDA13]) and deals with the potential speedup of a program using multiple processors compared to a single processor. Consider a program running on a single processor such that a fraction $(1 - f)$ of the execution time involves code that is inherently sequential, and a fraction f that involves code that is infinitely parallelizable with no scheduling overhead. Let T be the total execution time of the program using a single processor. Then the speedup using a parallel processor with N processors that fully exploits the parallel portion of the program is as follows:

$$\text{Speedup} = \frac{\text{Time to execute program on a single processor}}{\text{Time to execute program on } N \text{ parallel processors}}$$

$$= \frac{T(1-f) + Tf}{T(1-f) + \frac{Tf}{N}} = \frac{1}{(1-f) + \frac{f}{N}}$$

This equation is illustrated in Figures 2.3 and 2.4. Two important conclusions can be drawn:

1. When f is small, the use of parallel processors has little effect.
2. As N approaches infinity, speedup is bound by $1/(1 - f)$, so that there are diminishing returns for using more processors.

These conclusions are too pessimistic, an assertion first put forward in [GUST88]. For example, a server can maintain multiple threads or multiple tasks to handle multiple clients and execute the threads or tasks in parallel up to the limit of the number of processors. Many database applications involve computations on massive amounts of data that can be split up into multiple parallel tasks.

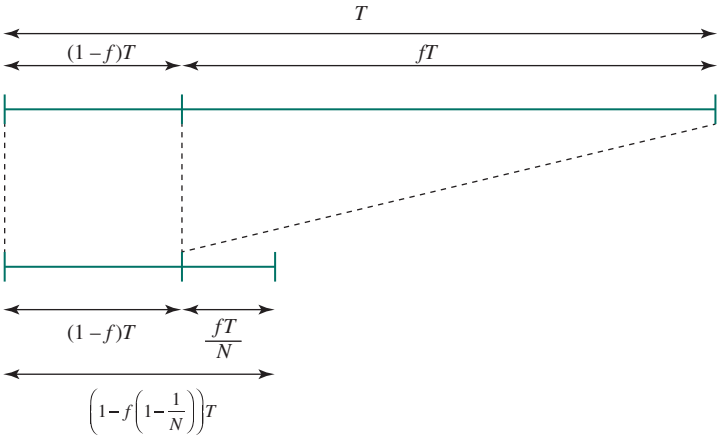


Figure 2.3 Illustration of Amdahl's Law

Nevertheless, Amdahl's law illustrates the problems facing industry in the development of multicore machines with an ever-growing number of cores: The software that runs on such machines must be adapted to a highly parallel execution environment to exploit the power of parallel processing.

Amdahl's law can be generalized to evaluate any design or technical improvement in a computer system. Consider any enhancement to a feature of a system that results in a speedup. The speedup can be expressed as

$$\text{Speedup} = \frac{\text{Performance after enhancement}}{\text{Performance before enhancement}} = \frac{\text{Execution time before enhancement}}{\text{Execution time after enhancement}} \tag{2.1}$$

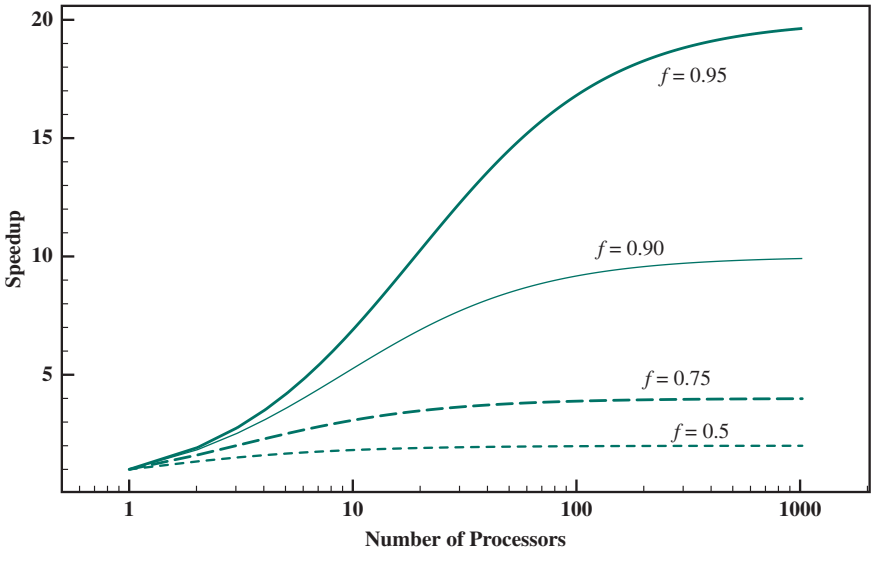


Figure 2.4 Amdahl's Law for Multiprocessors

Suppose that a feature of the system is used during execution a fraction of the time f, before enhancement, and that the speedup of that feature after enhancement is SU_f. Then the overall speedup of the system is

$$\text{Speedup} = \frac{1}{(1 - f) + \frac{f}{SU_f}}$$

EXAMPLE 2.1 Suppose that a task makes extensive use of floating-point operations, with 40% of the time consumed by floating-point operations. With a new hardware design, the floating-point module is sped up by a factor of K. Then the overall speedup is as follows:

$$\text{Speedup} = \frac{1}{0.6 + \frac{0.4}{K}}$$

Thus, independent of K, the maximum speedup is 1.67.

Little's Law

A fundamental and simple relation with broad applications is Little's Law [LITT61, LITT11].[4] We can apply it to almost any system that is statistically in steady state, and in which there is no leakage. Specifically, we have a steady state system to which items arrive at an average rate of λ items per unit time. The items stay in the system an average of W units of time. Finally, there is an average of L units in the system at any one time. Little's Law relates these three variables as $L = \lambda W$.

Using queuing theory terminology, Little's Law applies to a queuing system. The central element of the system is a server, which provides some service to items. Items from some population of items arrive at the system to be served. If the server is idle, an item is served immediately. Otherwise, an arriving item joins a waiting line, or queue. There can be a single queue for a single server, a single queue for multiple servers, or multiples queues, one for each of multiple servers. When a server has completed serving an item, the item departs. If there are items waiting in the queue, one is immediately dispatched to the server. The server in this model can represent anything that performs some function or service for a collection of items. Examples: A processor provides service to processes; a transmission line provides a transmission service to packets or frames of data; and an I/O device provides a read or write service for I/O requests.

To understand Little's formula, consider the following argument, which focuses on the experience of a single item. When the item arrives, it will find on

[4]The second reference is a retrospective article on his law that Little wrote 50 years after his original paper. That must be unique in the history of the technical literature, although Amdahl comes close, with a 46-year gap between [AMDA67] and [AMDA13].

average L items ahead of it, one being serviced and the rest in the queue. When the item leaves the system after being serviced, it will leave behind on average the same number of items in the system, namely L, because L is defined as the average number of items waiting. Further, the average time that the item was in the system was W. Since items arrive at a rate of λ, we can reason that in the time W, a total of λW items must have arrived. Thus $w = \lambda W$.

To summarize, under steady state conditions, the average number of items in a queuing system equals the average rate at which items arrive multiplied by the average time that an item spends in the system. This relationship requires very few assumptions. We do not need to know what the service time distribution is, what the distribution of arrival times is, or the order or priority in which items are served. Because of its simplicity and generality, Little's Law is extremely useful and has experienced somewhat of a revival due to the interest in performance problems related to multicore computers.

A very simple example, from [LITT11], illustrates how Little's Law might be applied. Consider a multicore system, with each core supporting multiple threads of execution. At some level, the cores share a common memory. The cores share a common main memory and typically share a common cache memory as well. In any case, when a thread is executing, it may arrive at a point at which it must retrieve a piece of data from the common memory. The thread stops and sends out a request for that data. All such stopped threads are in a queue. If the system is being used as a server, an analyst can determine the demand on the system in terms of the rate of user requests, and then translate that into the rate of requests for data from the threads generated to respond to an individual user request. For this purpose, each user request is broken down into subtasks that are implemented as threads. We then have λ = the average rate of total thread processing required after all members' requests have been broken down into whatever detailed subtasks are required. Define L as the average number of stopped threads waiting during some relevant time. Then W = average response time. This simple model can serve as a guide to designers as to whether user requirements are being met and, if not, provide a quantitative measure of the amount of improvement needed.

2.4 BASIC MEASURES OF COMPUTER PERFORMANCE

In evaluating processor hardware and setting requirements for new systems, performance is one of the key parameters to consider, along with cost, size, security, reliability, and, in some cases, power consumption.

It is difficult to make meaningful performance comparisons among different processors, even among processors in the same family. Raw speed is far less important than how a processor performs when executing a given application. Unfortunately, application performance depends not just on the raw speed of the processor but also on the instruction set, choice of implementation language, efficiency of the compiler, and skill of the programming done to implement the application.

In this section, we look at some traditional measures of processor speed. In the next section, we examine benchmarking, which is the most common approach to assessing processor and computer system performance. The following section discusses how to average results from multiple tests.

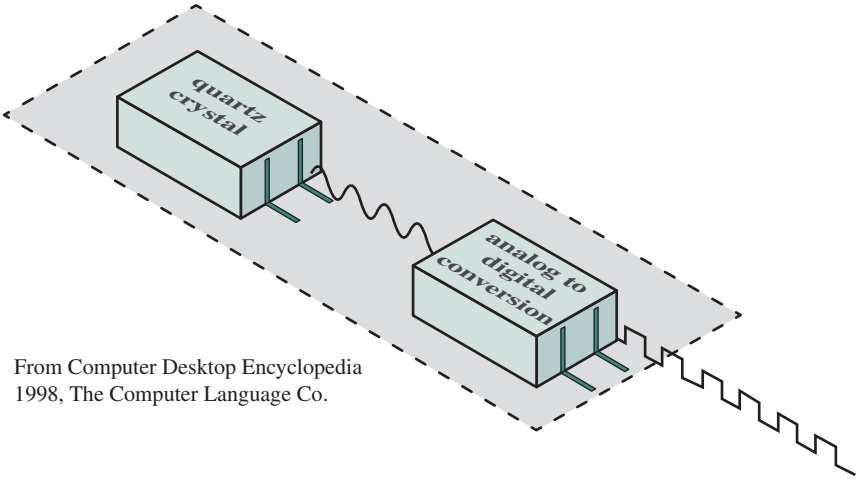
Clock Speed

Operations performed by a processor, such as fetching an instruction, decoding the instruction, performing an arithmetic operation, and so on, are governed by a system clock. Typically, all operations begin with the pulse of the clock. Thus, at the most fundamental level, the speed of a processor is dictated by the pulse frequency produced by the clock, measured in cycles per second, or Hertz (Hz).

Typically, clock signals are generated by a quartz crystal, which generates a constant sine wave while power is applied. This wave is converted into a digital voltage pulse stream that is provided in a constant flow to the processor circuitry (Figure 2.5). For example, a 1-GHz processor receives 1 billion pulses per second. The rate of pulses is known as the **clock rate**, or **clock speed**. One increment, or pulse, of the clock is referred to as a **clock cycle**, or a **clock tick**. The time between pulses is the **cycle time**.

The clock rate is not arbitrary, but must be appropriate for the physical layout of the processor. Actions in the processor require signals to be sent from one processor element to another. When a signal is placed on a line inside the processor, it takes some finite amount of time for the voltage levels to settle down so that an accurate value (logical 1 or 0) is available. Furthermore, depending on the physical layout of the processor circuits, some signals may change more rapidly than others. Thus, operations must be synchronized and paced so that the proper electrical signal (voltage) values are available for each operation.

The execution of an instruction involves a number of discrete steps, such as fetching the instruction from memory, decoding the various portions of the instruction, loading and storing data, and performing arithmetic and logical operations. Thus, most instructions on most processors require multiple clock cycles to complete. Some instructions may take only a few cycles, while others require dozens. In addition, when pipelining is used, multiple instructions are being executed simultaneously. Thus, a straight comparison of clock speeds on different processors does not tell the whole story about performance.



From Computer Desktop Encyclopedia
1998, The Computer Language Co.

Figure 2.5 System Clock

Instruction Execution Rate

A processor is driven by a clock with a constant frequency f or, equivalently, a constant cycle time τ, where $\tau = 1/f$. Define the instruction count, I_c, for a program as the number of machine instructions executed for that program until it runs to completion or for some defined time interval. Note that this is the number of instruction executions, not the number of instructions in the object code of the program. An important parameter is the average cycles per instruction (*CPI*) for a program. If all instructions required the same number of clock cycles, then *CPI* would be a constant value for a processor. However, on any given processor, the number of clock cycles required varies for different types of instructions, such as load, store, branch, and so on. Let CPI_i be the number of cycles required for instruction type i, and I_i be the number of executed instructions of type i for a given program. Then we can calculate an overall *CPI* as follows:

$$CPI = \frac{\sum_{i=1}^{n}(CPI_i \times I_i)}{I_c} \tag{2.2}$$

The processor time T needed to execute a given program can be expressed as

$$T = I_c \times CPI \times \tau$$

We can refine this formulation by recognizing that during the execution of an instruction, part of the work is done by the processor, and part of the time a word is being transferred to or from memory. In this latter case, the time to transfer depends on the memory cycle time, which may be greater than the processor cycle time. We can rewrite the preceding equation as

$$T = I_c \times [p + (m \times k)] \times \tau$$

where p is the number of processor cycles needed to decode and execute the instruction, m is the number of memory references needed, and k is the ratio between memory cycle time and processor cycle time. The five performance factors in the preceding equation (I_c, p, m, k, τ) are influenced by four system attributes: the design of the instruction set (known as *instruction set architecture*); compiler technology (how effective the compiler is in producing an efficient machine language program from a high-level language program); processor implementation; and cache and memory hierarchy. Table 2.1 is a matrix in which one dimension shows the five performance factors and the other dimension shows the four system attributes. An X in a cell indicates a system attribute that affects a performance factor.

Table 2.1 Performance Factors and System Attributes

	I_c	p	m	k	τ
Instruction set architecture	X	X			
Compiler technology	X	X	X		
Processor implementation		X			X
Cache and memory hierarchy				X	X

A common measure of performance for a processor is the rate at which instructions are executed, expressed as millions of instructions per second (MIPS), referred to as the **MIPS rate**. We can express the MIPS rate in terms of the clock rate and *CPI* as follows:

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6} \tag{2.3}$$

EXAMPLE 2.2 Consider the execution of a program that results in the execution of 2 million instructions on a 400-MHz processor. The program consists of four major types of instructions. The instruction mix and the *CPI* for each instruction type are given below, based on the result of a program trace experiment:

Instruction Type	*CPI*	Instruction Mix (%)
Arithmetic and logic	1	60
Load/store with cache hit	2	18
Branch	4	12
Memory reference with cache miss	8	10

The average *CPI* when the program is executed on a uniprocessor with the above trace results is $CPI = 0.6 + (2 \times 0.18) + (4 \times 0.12) + (8 \times 0.1) = 2.24$. The corresponding MIPS rate is $(400 \times 10^6)/(2.24 \times 10^6) \approx 178$.

Another common performance measure deals only with floating-point instructions. These are common in many scientific and game applications. Floating-point performance is expressed as millions of floating-point operations per second (MFLOPS), defined as follows:

$$\text{MFLOPS rate} = \frac{\text{Number of executed floating} - \text{point operations in a program}}{\text{Execution time} \times 10^6}$$

2.5 CALCULATING THE MEAN

In evaluating some aspect of computer system performance, it is often the case that a single number, such as execution time or memory consumed, is used to characterize performance and to compare systems. Clearly, a single number can provide only a very simplified view of a system's capability. Nevertheless, and especially in the field of benchmarking, single numbers are typically used for performance comparison [SMIT88].

As is discussed in Section 2.6, the use of benchmarks to compare systems involves calculating the mean value of a set of data points related to execution time. It turns out that there are multiple alternative algorithms that can be used for calculating a mean value, and this has been the source of some controversy in

the benchmarking field. In this section, we define these alternative algorithms and comment on some of their properties. This prepares us for a discussion in the next section of mean calculation in benchmarking.

The three common formulas used for calculating a mean are arithmetic, geometric, and harmonic. Given a set of n real numbers $(x_1, x_2, \ldots, x_n)$, the three means are defined as follows:

Arithmetic mean

$$AM = \frac{x_1 + \cdots + x_n}{n} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.4}$$

Geometric mean

$$GM = \sqrt[n]{x_1 \times \cdots \times x_n} = \left(\prod_{i=1}^{n} x_i\right)^{1/n} = e^{\left(\frac{1}{n}\sum_{i=1}^{n} \ln(x_i)\right)} \tag{2.5}$$

Harmonic mean

$$HM = \frac{n}{\left(\frac{1}{x_1}\right) + \cdots + \left(\frac{1}{x_n}\right)} = \frac{n}{\sum_{i=1}^{n}\left(\frac{1}{x_i}\right)} \qquad x_i > 0 \tag{2.6}$$

It can be shown that the following inequality holds:

$$AM \leq GM \leq HM$$

The values are equal only if $x_1 = x_2 = \ldots x_n$.

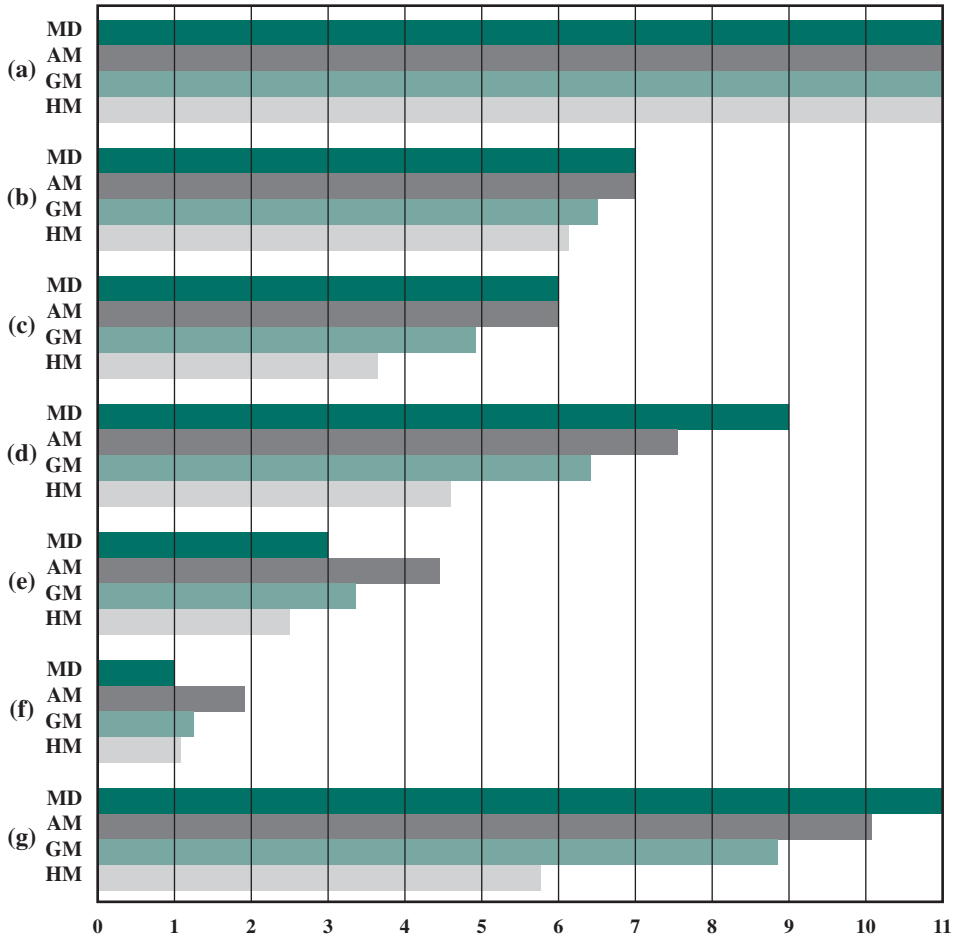
We can get a useful insight into these alternative calculations by defining the functional mean. Let $f(x)$ be a continuous monotonic function defined in the interval $0 \leq y < \infty$. The functional mean with respect to the function $f(x)$ for n positive real numbers $(x_1, x_2, \ldots, x_n)$ is defined as

Functional mean $$FM = f^{-1}\left(\frac{f(x_1) + \cdots + f(x_n)}{n}\right) = f^{-1}\left(\frac{1}{n}\sum_{i=1}^{n} f(x_i)\right)$$

where $f^{-1}(x)$ is the inverse of $f(x)$. The mean values defined in Equations (2.1) through (2.3) are special cases of the functional mean, as follows:

- AM is the FM with respect to $f(x) = x$
- GM is the FM with respect to $f(x) = \ln x$
- HM is the FM with respect to $f(x) = 1/x$

EXAMPLE 2.3 Figure 2.6 illustrates the three means applied to various data sets, each of which has eleven data points and a maximum data point value of 11. The median value is also included in the chart. Perhaps what stands out the most in this figure is that the HM has a tendency to produce a misleading result when the data is skewed to larger values or when there is a small-value outlier.



(a) Constant (11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11)
(b) Clustered around a central value (3, 5, 6, 6, 7, 7, 7, 8, 8, 9, 11)
(c) Uniform distribution (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
(d) Large-number bias (1, 4, 4, 7, 7, 9, 9, 10, 10, 11, 11)
(e) Small-number bias(1, 1, 2, 2, 3, 3, 5, 5, 8, 8, 11)
(f) Upper outlier (11, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
(g) Lower outlier (1, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11)

MD = median
AM = arithmetic mean
GM = geometric mean
HM = harmonic mean

Figure 2.6 Comparison of Means on Various Data Sets (each set has a maximum data point value of 11)

Let us now consider which of these means are appropriate for a given performance measure. As a preface to these remarks, it should be noted that a number of papers ([CITR06], [FLEM86], [GILA95], [JACO95], [JOHN04], [MASH04], [SMIT88]) and books ([HENN12], [HWAN93], [JAIN91], [LILJ00]) over the years have argued the pros and cons of the three means for performance analysis and come to conflicting conclusions. To simplify a complex controversy, we just note that the conclusions reached depend very much on the examples chosen and the way in which the objectives are stated.

Arithmetic Mean

An AM is an appropriate measure if the sum of all the measurements is a meaningful and interesting value. The AM is a good candidate for comparing the execution time performance of several systems. For example, suppose we were interested in using a system for large-scale simulation studies and wanted to evaluate several alternative products. On each system we could run the simulation multiple times with different input values for each run, and then take the average execution time across all runs. The use of multiple runs with different inputs should ensure that the results are not heavily biased by some unusual feature of a given input set. The AM of all the runs is a good measure of the system's performance on simulations, and a good number to use for system comparison.

The AM used for a time-based variable (e.g., seconds), such as program execution time, has the important property that it is directly proportional to the total time. So, if the total time doubles, the mean value doubles.

Harmonic Mean

For some situations, a system's execution rate may be viewed as a more useful measure of the value of the system. This could be either the instruction execution rate, measured in MIPS or MFLOPS, or a program execution rate, which measures the rate at which a given type of program can be executed. Consider how we wish the calculated mean to behave. It makes no sense to say that we would like the mean rate to be proportional to the total rate, where the total rate is defined as the sum of the individual rates. The sum of the rates would be a meaningless statistic. Rather, we would like the mean to be inversely proportional to the total execution time. For example, if the total time to execute all the benchmark programs in a suite of programs is twice as much for system C as for system D, we would want the mean value of the execution rate to be half as much for system C as for system D.

Let us look at a basic example and first examine how the AM performs. Suppose we have a set of n benchmark programs and record the execution times of each program on a given system as t_1, t_2, ..., t_n. For simplicity, let us assume that each program executes the same number of operations Z; we could weight the individual programs and calculate accordingly but this would not change the conclusion of our argument. The execution rate for each individual program is $R_i = Z/t_i$. We use the AM to calculate the average execution rate.

$$AM = \frac{1}{n}\sum_{i=1}^{n} R_i = \frac{1}{n}\sum_{i=1}^{n} \frac{Z}{t_i} = \frac{Z}{n}\sum_{i=1}^{n} \frac{1}{t_i}$$

We see that the AM execution rate is proportional to the sum of the inverse execution times, which is not the same as being inversely proportional to the sum of the execution times. Thus, the AM does not have the desired property.

The HM yields the following result.

$$HM = \frac{n}{\sum_{i=1}^{n}\left(\frac{1}{R_i}\right)} = \frac{n}{\sum_{i=1}^{n}\left(\frac{1}{Z/t_i}\right)} = \frac{nZ}{\sum_{i=1}^{n} t_i}$$

The HM is inversely proportional to the total execution time, which is the desired property.

EXAMPLE 2.4 A simple numerical example will illustrate the difference between the two means in calculating a mean value of the rates, shown in Table 2.2. The table compares the performance of three computers on the execution of two programs. For simplicity, we assume that the execution of each program results in the execution of 10^8 floating-point operations. The left half of the table shows the execution times for each computer running each program, the total execution time, and the AM of the execution times. Computer A executes in less total time than B, which executes in less total time than C, and this is reflected accurately in the AM.

The right half of the table provides a comparison in terms of rates, expressed in MFLOPS. The rate calculation is straightforward. For example, program 1 executes 100 million floating-point operations. Computer A takes 2 seconds to execute the program for a MFLOPS rate of $100/2 = 50$. Next, consider the AM of the rates. The greatest value is for computer A, which suggests that A is the fastest computer. In terms of total execution time, A has the minimum time, so it is the fastest computer of the three. But the AM of rates shows B as slower than C, whereas in fact B is faster than C. Looking at the HM values, we see that they correctly reflect the speed ordering of the computers. This confirms that the HM is preferred when calculating rates.

The reader may wonder why go through all this effort. If we want to compare execution times, we could simply compare the total execution times of the three systems. If we want to compare rates, we could simply take the inverse of the total execution time, as shown in the table. There are two reasons for doing the individual calculations rather than only looking at the aggregate numbers:

Table 2.2 A Comparison of Arithmetic and Harmonic Means for Rates

	Computer A time (secs)	Computer B time (secs)	Computer C time (secs)	Computer A rate (MFLOPS)	Computer B rate (MFLOPS)	Computer C rate (MFLOPS)
Program 1 (10^8 FP ops)	2.0	1.0	0.75	50	100	133.33
Program 2 (10^8 FP ops)	0.75	2.0	4.0	133.33	50	25
Total execution time	2.75	3.0	4.75	—	—	—
Arithmetic mean of times	1.38	1.5	2.38	—	—	—
Inverse of total execution time (1/sec)	0.36	0.33	0.21	—	—	—
Arithmetic mean of rates	—	—	—	91.67	75.00	79.17
Harmonic mean of rates	—	—	—	72.72	66.67	42.11

1. A customer or researcher may be interested not only in the overall average performance but also performance against different types of benchmark programs, such as business applications, scientific modeling, multimedia applications, and systems programs. Thus, a breakdown by type of benchmark is needed as well as a total.
2. Usually, the different programs used for evaluation are weighted differently. In Table 2.2, it is assumed that the two test programs execute the same number of operations. If that is not the case, we may want to weight accordingly. Or different programs could be weighted differently to reflect importance or priority.

Let us see what the result is if test programs are weighted proportional to the number of operations. Following the preceding notation, each program i executes Zi instructions in a time ti. Each rate is weighted by the instructions count. The weighted HM is therefore:

$$WHM = \frac{1}{\sum_{i=1}^{n}\left(\left(\frac{Z_i}{\sum_{j=1}^{n} Z_j}\right)\left(\frac{1}{R_i}\right)\right)} = \frac{n}{\sum_{i=1}^{n}\left(\left(\frac{Z_i}{\sum_{j=1}^{n} Z_j}\right)\left(\frac{t_i}{Z_i}\right)\right)} = \frac{\sum_{j=1}^{n} Z_j}{\sum_{i=1}^{n} t_i} \quad (2.7)$$

We see that the weighted HM is the quotient of the sum of the operation count divided by the sum of the execution times.

Geometric Mean

Looking at the equations for the three types of means, it is easier to get an intuitive sense of the behavior of the AM and the HM than that of the GM. Several observations, from [FEIT15], may be helpful in this regard. First, we note that with respect to changes in values, the GM gives equal weight to all of the values in the data set. For example, suppose the set of data values to be averaged includes a few large values and more small values. Here, the AM is dominated by the large values. A change of 10% in the largest value will have a noticeable effect, while a change in the smallest value by the same factor will have a negligible effect. In contrast, a change in value by 10% of any of the data values results in the same change in the GM: $\sqrt[n]{1.1}$.

EXAMPLE 2.5 This point is illustrated by data set (e) in Figure 2.6. Here are the effects of increasing either the maximum or the minimum value in the data set by 10%:

	Geometric Mean	Arithmetic Mean
Original value	3.37	4.45
Increase max value from 11 to 12.1 (+10%)	3.40 (+ 0.87%)	4.55 (+ 2.24%)
Increase min value from 1 to 1.1 (+10%)	3.40 (+ 0.87%)	4.46 (+ 0.20%)

A second observation is that for the GM of a ratio, the GM of the ratios equals the ratio of the GMs:

$$GM = \left(\prod_{i=1}^{n} \frac{Z_i}{t_i}\right)^{1/n} = \frac{\left(\prod_{i=1}^{n} Z_i\right)^{1/n}}{\left(\prod_{i=1}^{n} t_i\right)^{1/n}} \tag{2.8}$$

Compare this with Equation 2.4.

For use with execution times, as opposed to rates, one drawback of the GM is that it may be non-monotonic relative to the more intuitive AM. In other words there may be cases where the AM of one data set is larger than that of another set, but the GM is smaller.

EXAMPLE 2.6 In Figure 2.6, the AM for data set d is larger than the AM for data set c, but the opposite is true for the GM.

	Data set c	Data set d
Arithmetic mean	7.00	7.55
Geometric mean	6.68	6.42

One property of the GM that has made it appealing for benchmark analysis is that it provides consistent results when measuring the relative performance of machines. This is in fact what benchmarks are primarily used for: to compare one machine with another in terms of performance metrics. The results, as we have seen, are expressed in terms of values that are normalized to a reference machine.

EXAMPLE 2.7 A simple example will illustrate the way in which the GM exhibits consistency for normalized results. In Table 2.3, we use the same performance results as were used in Table 2.2. In Table 2.3a, all results are normalized to Computer A, and the means are calculated on the normalized values. Based on total execution time, A is faster than B, which is faster than C. Both the AMs and GMs of the normalized times reflect this. In Table 2.3b, the systems are now normalized to B. Again the GMs correctly reflect the relative speeds of the three computers, but now the AM produces a different ordering.

Sadly, consistency does not always produce correct results. In Table 2.4, some of the execution times are altered. Once again, the AM reports conflicting results for the two normalizations. The GM reports consistent results, but the result is that B is faster than A and C, which are equal.

It is examples like this that have fueled the "benchmark means wars" in the citations listed earlier. It is safe to say that no single number can provide all the information that one needs for comparing performance across systems. However,

Table 2.3 A Comparison of Arithmetic and Geometric Means for Normalized Results

(a) Results normalized to Computer A

	Computer A time	Computer B time	Computer C time
Program 1	2.0 (1.0)	1.0 (0.5)	0.75 (0.38)
Program 2	0.75 (1.0)	2.0 (2.67)	4.0 (5.33)
Total execution time	2.75	3.0	4.75
Arithmetic mean of normalized times	1.00	1.58	2.85
Geometric mean of normalized times	1.00	1.15	1.41

(b) Results normalized to Computer B

	Computer A time	Computer B time	Computer C time
Program 1	2.0 (2.0)	1.0 (1.0)	0.75 (0.75)
Program 2	0.75 (0.38)	2.0 (1.0)	4.0 (2.0)
Total execution time	2.75	3.0	4.75
Arithmetic mean of normalized times	1.19	1.00	1.38
Geometric mean of normalized times	0.87	1.00	1.22

Table 2.4 Another Comparison of Arithmetic and Geometric Means for Normalized Results

(a) Results normalized to Computer A

	Computer A time	Computer B time	Computer C time
Program 1	2.0 (1.0)	1.0 (0.5)	0.20 (0.1)
Program 2	0.4 (1.0)	2.0 (5.0)	4.0 (10.0)
Total execution time	2.4	3.00	4.2
Arithmetic mean of normalized times	1.00	2.75	5.05
Geometric mean of normalized times	1.00	1.58	1.00

(b) Results normalized to Computer B

	Computer A time	Computer B time	Computer C time
Program 1	2.0 (2.0)	1.0 (1.0)	0.20 (0.2)
Program 2	0.4 (0.2)	2.0 (1.0)	4.0 (2.0)
Total execution time	2.4	3.00	4.2
Arithmetic mean of normalized times	1.10	1.00	1.10
Geometric mean of normalized times	0.63	1.00	0.63

despite the conflicting opinions in the literature, SPEC has chosen to use the GM, for several reasons:

1. As mentioned, the GM gives consistent results regardless of which system is used as a reference. Because benchmarking is primarily a comparison analysis, this is an important feature.
2. As documented in [MCMA93], and confirmed in subsequent analyses by SPEC analysts [MASH04], the GM is less biased by outliers than the HM or AM.
3. [MASH04] demonstrates that distributions of performance ratios are better modeled by lognormal distributions than by normal ones, because of the generally skewed distribution of the normalized numbers. This is confirmed in [CITR06]. And, as shown in Equation (2.5), the GM can be described as the back-transformed average of a lognormal distribution.

2.6 BENCHMARKS AND SPEC

Benchmark Principles

Measures such as MIPS and MFLOPS have proven inadequate to evaluating the performance of processors. Because of differences in instruction sets, the instruction execution rate is not a valid means of comparing the performance of different architectures.

EXAMPLE 2.8 Consider this high-level language statement:

```
A = B + C   /* assume all quantities in main memory */
```

With a traditional instruction set architecture, referred to as a complex instruction set computer (CISC), this instruction can be compiled into one processor instruction:

```
add   mem(B), mem(C), mem (A)
```

On a typical RISC machine, the compilation would look something like this:

```
load  mem(B), reg(1);
load  mem(C), reg(2);
add   reg(1), reg(2), reg(3);
store reg(3), mem (A)
```

Because of the nature of the RISC architecture (discussed in Chapter 15), both machines may execute the original high-level language instruction in about the same time. If this example is representative of the two machines, then if the CISC machine is rated at 1 MIPS, the RISC machine would be rated at 4 MIPS. But both do the same amount of high-level language work in the same amount of time.

Another consideration is that the performance of a given processor on a given program may not be useful in determining how that processor will perform on a very different type of application. Accordingly, beginning in the late 1980s and early 1990s, industry and academic interest shifted to measuring the performance of

systems using a set of benchmark programs. The same set of programs can be run on different machines and the execution times compared. Benchmarks provide guidance to customers trying to decide which system to buy, and can be useful to vendors and designers in determining how to design systems to meet benchmark goals.

[WEIC90] lists the following as desirable characteristics of a benchmark program:

1. It is written in a high-level language, making it portable across different machines.
2. It is representative of a particular kind of programming domain or paradigm, such as systems programming, numerical programming, or commercial programming.
3. It can be measured easily.
4. It has wide distribution.

SPEC Benchmarks

The common need in industry and academic and research communities for generally accepted computer performance measurements has led to the development of standardized benchmark suites. A benchmark suite is a collection of programs, defined in a high-level language, that together attempt to provide a representative test of a computer in a particular application or system programming area. The best known such collection of benchmark suites is defined and maintained by the Standard Performance Evaluation Corporation (SPEC), an industry consortium. This organization defines several benchmark suites aimed at evaluating computer systems. SPEC performance measurements are widely used for comparison and research purposes.

The best known of the SPEC benchmark suites is SPEC CPU2006. This is the industry standard suite for processor-intensive applications. That is, SPEC CPU2006 is appropriate for measuring performance for applications that spend most of their time doing computation rather than I/O.

Other SPEC suites include the following:

- **SPECviewperf:** Standard for measuring 3D graphics performance based on professional applications.
- **SPECwpc:** benchmark to measure all key aspects of workstation performance based on diverse professional applications, including media and entertainment, product development, life sciences, financial services, and energy.
- **SPECjvm2008:** Intended to evaluate performance of the combined hardware and software aspects of the Java Virtual Machine (JVM) client platform.
- **SPECjbb2013 (Java Business Benchmark):** A benchmark for evaluating server-side Java-based electronic commerce applications.
- **SPECsfs2008:** Designed to evaluate the speed and request-handling capabilities of file servers.
- **SPECvirt_sc2013:** Performance evaluation of datacenter servers used in virtualized server consolidation. Measures the end-to-end performance of all system components including the hardware, virtualization platform, and the virtualized guest operating system and application software. The benchmark supports hardware virtualization, operating system virtualization, and hardware partitioning schemes.

The CPU2006 suite is based on existing applications that have already been ported to a wide variety of platforms by SPEC industry members. In order to make the benchmark results reliable and realistic, the CPU2006 benchmarks are drawn from real-life applications, rather than using artificial loop programs or synthetic benchmarks. The suite consists of 12 integer benchmarks written in C and C++, and 17 floating-point benchmarks written in C, C++, and Fortran (Tables 2.5 and 2.6). The suite contains over 3 million lines of code. This is the fifth generation of

Table 2.5 SPEC CPU2006 Integer Benchmarks

Benchmark	Reference time (hours)	Instr count (billion)	Language	Application Area	Brief Description
400.perlbench	2.71	2378	C	Programming Language	PERL programming language interpreter, applied to a set of three programs.
401.bzip2	2.68	2472	C	Compression	General-purpose data compression with most work done in memory, rather than doing I/O.
403.gcc	2.24	1064	C	C Compiler	Based on gcc Version 3.2, generates code for Opteron.
429.mcf	2.53	327	C	Combinatorial Optimization	Vehicle scheduling algorithm.
445.gobmk	2.91	1603	C	Artificial Intelligence	Plays the game of Go, a simply described but deeply complex game.
456.hmmer	2.59	3363	C	Search Gene Sequence	Protein sequence analysis using profile-hidden Markov models.
458.sjeng	3.36	2383	C	Artificial Intelligence	A highly ranked chess program that also plays several chess variants.
462.libquantum	5.76	3555	C	Physics / Quantum Computing	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.
464.h264ref	6.15	3731	C	Video Compression	H.264/AVC (Advanced Video Coding) video compression.
471.omnetpp	1.74	687	C++	Discrete Event Simulation	Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.
473.astar	1.95	1200	C++	Path-finding Algorithms	Pathfinding library for 2D maps.
483.xalancbmk	1.92	1184	C++	XML Processing	A modified version of Xalan-C++, which transforms XML documents to other document types.

Table 2.6 SPEC CPU2006 Floating-Point Benchmarks

Benchmark	Reference time (hours)	Instr count (billion)	Language	Application Area	Brief Description
410.bwaves	3.78	1176	Fortran	Fluid Dynamics	Computes 3D transonic transient laminar viscous flow.
416.gamess	5.44	5189	Fortran	Quantum Chemistry	Quantum chemical computations.
433.milc	2.55	937	C	Physics / Quantum Chromody-namics	Simulates behavior of quarks and gluons.
434.zeusmp	2.53	1566	Fortran	Physics / CFD	Computational fluid dynamics simulation of astrophysical phenomena.
435.gromacs	1.98	1958	C, Fortran	Biochemistry / Molecular Dynamics	Simulates Newtonian equations of motion for hundreds to millions of particles.
436. cactusADM	3.32	1376	C, Fortran	Physics / General Relativity	Solves the Einstein evolu-tion equations.
437.leslie3d	2.61	1273	Fortran	Fluid Dynamics	Models fuel injection flows.
444.namd	2.23	2483	C++	Biology / Molecular Dynamics	Simulates large biomolecu-lar systems.
447.dealII	3.18	2323	C++	Finite Element Analysis	Program library targeted at adaptive finite elements and error estimation.
450.soplex	2.32	703	C++	Linear Pro-gramming, Optimization	Test cases include railroad planning and military airlift models.
453.povray	1.48	940	C++	Image Ray-Tracing	3D image rendering.
454.calculix	2.29	3,04	C, Fortran	Structural Mechanics	Finite element code for linear and nonlinear 3D structural applications.
459. GemsFDTD	2.95	1320	Fortran	Computa-tional Elec-tromagnetics	Solves the Maxwell equa-tions in 3D.
465.tonto	2.73	2392	Fortran	Quantum Chemistry	Quantum chemistry pack-age, adapted for crystallo-graphic tasks.
470.lbm	3.82	1500	C	Fluid Dynamics	Simulates incompressible fluids in 3D.
481.wrf	3.10	1684	C, Fortran	Weather	Weather forecasting model.
482.sphinx3	5.41	2472	C	Speech Recognition	Speech recognition software.

processor-intensive suites from SPEC, replacing SPEC CPU2000, SPEC CPU95, SPEC CPU92, and SPEC CPU89 [HENN07].

To better understand published results of a system using CPU2006, we define the following terms used in the SPEC documentation:

- **Benchmark:** A program written in a high-level language that can be compiled and executed on any computer that implements the compiler.
- **System under test:** This is the system to be evaluated.
- **Reference machine:** This is a system used by SPEC to establish a baseline performance for all benchmarks. Each benchmark is run and measured on this machine to establish a reference time for that benchmark. A system under test is evaluated by running the CPU2006 benchmarks and comparing the results for running the same programs on the reference machine.
- **Base metric:** These are required for all reported results and have strict guidelines for compilation. In essence, the standard compiler with more or less default settings should be used on each system under test to achieve comparable results.
- **Peak metric:** This enables users to attempt to optimize system performance by optimizing the compiler output. For example, different compiler options may be used on each benchmark, and feedback-directed optimization is allowed.
- **Speed metric:** This is simply a measurement of the time it takes to execute a compiled benchmark. The speed metric is used for comparing the ability of a computer to complete single tasks.
- **Rate metric:** This is a measurement of how many tasks a computer can accomplish in a certain amount of time; this is called a **throughput**, capacity, or rate measure. The rate metric allows the system under test to execute simultaneous tasks to take advantage of multiple processors.

SPEC uses a historical Sun system, the "Ultra Enterprise 2," which was introduced in 1997, as the reference machine. The reference machine uses a 296-MHz UltraSPARC II processor. It takes about 12 days to do a rule-conforming run of the base metrics for CINT2006 and CFP2006 on the CPU2006 reference machine. Tables 2.5 and 2.6 show the amount of time to run each benchmark using the reference machine. The tables also show the dynamic instruction counts on the reference machine, as reported in [PHAN07]. These values are the actual number of instructions executed during the run of each program.

We now consider the specific calculations that are done to assess a system. We consider the integer benchmarks; the same procedures are used to create a floating-point benchmark value. For the integer benchmarks, there are 12 programs in the test suite. Calculation is a three-step process (Figure 2.7):

1. The first step in evaluating a system under test is to compile and run each program on the system three times. For each program, the runtime is measured and the median value is selected. The reason to use three runs and take the median value is to account for variations in execution time that are not intrinsic to the program, such as disk access time variations, and OS kernel execution variations from one run to another.

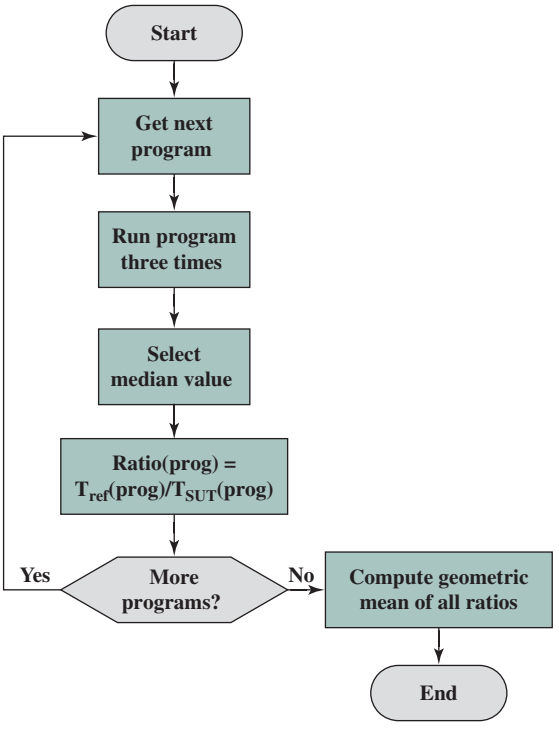


Figure 2.7 SPEC Evaluation Flowchart

2. Next, each of the 12 results is normalized by calculating the runtime ratio of the reference run time to the system run time. The ratio is calculated as follows:

$$r_i = \frac{Tref_i}{Tsut_i} \tag{2.9}$$

where $Tref_i$ is the execution time of benchmark program i on the reference system and $Tsut_i$ is the execution time of benchmark program i on the system under test. Thus, ratios are higher for faster machines.

3. Finally, the geometric mean of the 12 runtime ratios is calculated to yield the overall metric:

$$r_G = \left(\prod_{i=1}^{12} r_i\right)^{1/12}$$

For the integer benchmarks, four separate metrics can be calculated:

- **SPECint2006:** The geometric mean of 12 normalized ratios when the benchmarks are compiled with peak tuning.
- **SPECint_base2006:** The geometric mean of 12 normalized ratios when the benchmarks are compiled with base tuning.
- **SPECint_rate2006:** The geometric mean of 12 normalized throughput ratios when the benchmarks are compiled with peak tuning.
- **SPECint_rate_base2006:** The geometric mean of 12 normalized throughput ratios when the benchmarks are compiled with base tuning.

EXAMPLE 2.9 The results for the Sun Blade 1000 are shown in Table 2.7a. One of the SPEC CPU2006 integer benchmark is 464.h264ref. This is a reference implementation of H.264/AVC (Advanced Video Coding), the latest state-of-the-art video compression standard. The Sun Blade 1000 executes this program in a median time of 5,259 seconds. The reference implementation requires 22,130 seconds. The ratio is calculated as: 22,130/5,259 = 4.21. The speed metric is calculated by taking the twelfth root of the product of the ratios:

$$(3.18 \times 2.96 \times 2.98 \times 3.91 \times 3.17 \times 3.61 \times 3.51 \times 2.01 \times 4.21 \times 2.43 \times 2.75 \times 3.42)^{1/12} = 3.12$$

The rate metrics take into account a system with multiple processors. To test a machine, a number of copies N is selected—usually this is equal to the number of processors or the number of simultaneous threads of execution on the test system. Each individual test program's rate is determined by taking the median of three runs. Each run consists of N copies of the program running simultaneously on the test system. The execution time is the time it takes for all the copies to finish (i.e., the time from when the first copy starts until the last copy finishes). The rate metric for that program is calculated by the following formula:

$$rate_i = N \times \frac{Tref_i}{Tsut_i}$$

The rate score for the system under test is determined from a geometric mean of rates for each program in the test suite.

EXAMPLE 2.10 The results for the Sun Blade X6250 are shown in Table 2.7b. This system has two processor chips, with two cores per chip, for a total of four cores. To get the rate metric, each benchmark program is executed simultaneously on all four cores, with the execution time being the time from the start of all four copies to the end of the slowest run. The speed ratio is calculated as before, and the rate value is simply four times the speed ratio. The final rate metric is found by taking the geometric mean of the rate values:

$$(78.63 \times 62.97 \times 60.87 \times 77.29 \times 65.87 \times 83.68 \times 76.70 \times 134.98 \times 106.65 \times 40.39 \times 48.41 \times 65.40)^{1/12} = 71.59$$

Table 2.7 Some SPEC CINT2006 Results

(a) Sun Blade 1000

Benchmark	Execution time (secs)	Execution time (secs)	Execution time (secs)	Reference time (secs)	Ratio
400.perlbench	3077	3076	3080	9770	3.18
401.bzip2	3260	3263	3260	9650	2.96
403.gcc	2711	2701	2702	8050	2.98
429.mcf	2356	2331	2301	9120	3.91
445.gobmk	3319	3310	3308	10,490	3.17
456.hmmer	2586	2587	2601	9330	3.61

(*Continued*)

Table 2.7 (*Continued*)

(a) Sun Blade 1000

Benchmark	Execution time (secs)	Execution time (secs)	Execution time (secs)	Reference time (secs)	Ratio
458.sjeng	3452	3449	3449	12,100	3.51
462.libquantum	10,318	10,319	10,273	20,720	2.01
464.h264ref	5246	5290	5259	22,130	4.21
471.omnetpp	2565	2572	2582	6250	2.43
473.astar	2522	2554	2565	7020	2.75
483.xalancbmk	2014	2018	2018	6900	3.42

(b) Sun Blade X6250

Benchmark	Execution time (secs)	Execution time (secs)	Execution time (secs)	Reference time (secs)	Ratio	Rate
400.perlbench	497	497	497	9770	19.66	78.63
401.bzip2	613	614	613	9650	15.74	62.97
403.gcc	529	529	529	8050	15.22	60.87
429.mcf	472	472	473	9120	19.32	77.29
445.gobmk	637	637	637	10,490	16.47	65.87
456.hmmer	446	446	446	9330	20.92	83.68
458.sjeng	631	632	630	12,100	19.18	76.70
462.libquantum	614	614	614	20,720	33.75	134.98
464.h264ref	830	830	830	22,130	26.66	106.65
471.omnetpp	619	620	619	6250	10.10	40.39
473.astar	580	580	580	7020	12.10	48.41
483.xalancbmk	422	422	422	6900	16.35	65.40

2.7 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

Amdahl's law	functional mean (FM)	microprocessor
arithmetic mean (AM)	general-purpose computing on GPU (GPGPU)	MIPS rate
base metric	geometric mean (GM)	multicore
benchmark	graphics processing unit (GPU)	peak metric
clock cycle	harmonic mean (HM)	rate metric
clock cycle time	instruction execution rate	reference machine
clock rate	Little's law	speed metric
clock speed	many integrated core (MIC)	SPEC
clock tick		system under test
cycles per instruction (*CPI*)		throughput

Review Questions

2.1 List and briefly define some of the techniques used in contemporary processors to increase speed.
2.2 Explain the concept of performance balance.
2.3 Explain the differences among multicore systems, MICs, and GPGPUs.
2.4 Briefly characterize Amdahl's law.
2.5 Briefly characterize Little's law.
2.6 Define MIPS and FLOPS.
2.7 List and define three methods for calculating a mean value of a set of data values.
2.8 List the desirable characteristics of a benchmark program.
2.9 What are the SPEC benchmarks?
2.10 What are the differences among base metric, peak metric, speed metric, and rate metric?

Problems

2.1 A benchmark program is run on a 40 MHz processor. The executed program consists of 100,000 instruction executions, with the following instruction mix and clock cycle count:

Instruction Type	Instruction Count	Cycles per Instruction
Integer arithmetic	45,000	1
Data transfer	32,000	2
Floating point	15,000	2
Control transfer	8000	2

Determine the effective *CPI*, MIPS rate, and execution time for this program.

2.2 Consider two different machines, with two different instruction sets, both of which have a clock rate of 200 MHz. The following measurements are recorded on the two machines running a given set of benchmark programs:

Instruction Type	Instruction Count (millions)	Cycles per Instruction
Machine A		
Arithmetic and logic	8	1
Load and store	4	3
Branch	2	4
Others	4	3
Machine A		
Arithmetic and logic	10	1
Load and store	8	2
Branch	2	4
Others	4	3

a. Determine the effective *CPI*, MIPS rate, and execution time for each machine.
b. Comment on the results.

2.3 Early examples of CISC and RISC design are the VAX 11/780 and the IBM RS/6000, respectively. Using a typical benchmark program, the following machine characteristics result:

Processor	Clock Frequency (MHz)	Performance (MIPS)	CPU Time (secs)
VAX 11/780	5	1	12 x
IBM RS/6000	25	18	x

The final column shows that the VAX required 12 times longer than the IBM measured in CPU time.

a. What is the relative size of the instruction count of the machine code for this benchmark program running on the two machines?

b. What are the *CPI* values for the two machines?

2.4 Four benchmark programs are executed on three computers with the following results:

	Computer A	Computer B	Computer C
Program 1	1	10	20
Program 2	1000	100	20
Program 3	500	1000	50
Program 4	100	800	100

The table shows the execution time in seconds, with 100,000,000 instructions executed in each of the four programs. Calculate the MIPS values for each computer for each program. Then calculate the arithmetic and harmonic means assuming equal weights for the four programs, and rank the computers based on arithmetic mean and harmonic mean.

2.5 The following table, based on data reported in the literature [HEAT84], shows the execution times, in seconds, for five different benchmark programs on three machines.

Benchmark	Processor		
	R	M	Z
E	417	244	134
F	83	70	70
H	66	153	135
I	39,449	35,527	66,000
K	772	368	369

a. Compute the speed metric for each processor for each benchmark, normalized to machine R. That is, the ratio values for R are all 1.0. Other ratios are calculated using Equation (2.5) with R treated as the reference system. Then compute the arithmetic mean value for each system using Equation (2.3). This is the approach taken in [HEAT84].

b. Repeat part (a) using M as the reference machine. This calculation was not tried in [HEAT84].

c. Which machine is the slowest based on each of the preceding two calculations?

d. Repeat the calculations of parts (a) and (b) using the geometric mean, defined in Equation (2.6). Which machine is the slowest based on the two calculations?

2.6 To clarify the results of the preceding problem, we look at a simpler example.

Benchmark	Processor		
	X	Y	Z
1	20	10	40
2	40	80	20

a. Compute the arithmetic mean value for each system using X as the reference machine and then using Y as the reference machine. Argue that intuitively the three machines have roughly equivalent performance and that the arithmetic mean gives misleading results.

b. Compute the geometric mean value for each system using X as the reference machine and then using Y as the reference machine. Argue that the results are more realistic than with the arithmetic mean.

2.7 Consider the example in Section 2.5 for the calculation of average *CPI* and MIPS rate, which yielded the result of $CPI = 2.24$ and MIPS rate $= 178$. Now assume that the program can be executed in eight parallel tasks or threads with roughly equal number of instructions executed in each task. Execution is on an 8-core system with each core (processor) having the same performance as the single processor originally used. Coordination and synchronization between the parts adds an extra 25,000 instruction executions to each task. Assume the same instruction mix as in the example for each task, but increase the *CPI* for memory reference with cache miss to 12 cycles due to contention for memory.

a. Determine the average *CPI*.

b. Determine the corresponding MIPS rate.

c. Calculate the speedup factor.

d. Compare the actual speedup factor with the theoretical speedup factor determined by Amdhal's law.

2.8 A processor accesses main memory with an average access time of T_2. A smaller cache memory is interposed between the processor and main memory. The cache has a significantly faster access time of $T_1 < T_2$. The cache holds, at any time, copies of some main memory words and is designed so that the words more likely to be accessed in the near future are in the cache. Assume that the probability that the next word accessed by the processor is in the cache is H, known as the hit ratio.

a. For any single memory access, what is the theoretical speedup of accessing the word in the cache rather than in main memory?

b. Let T be the average access time. Express T as a function of T_1, T_2, and H. What is the overall speedup as a function of H?

c. In practice, a system may be designed so that the processor must first access the cache to determine if the word is in the cache and, if it is not, then access main memory, so that on a miss (opposite of a hit), memory access time is $T_1 + T_2$. Express T as a function of T_1, T_2, and H. Now calculate the speedup and compare to the result produced in part (b).

2.9 The owner of a shop observes that on average 18 customers per hour arrive and there are typically 8 customers in the shop. What is the average length of time each customer spends in the shop?

2.10 We can gain more insight into Little's law by considering Figure 2.8a. Over a period of time T, a total of C items arrive at a system, wait for service, and complete service. The upper solid line shows the time sequence of arrivals, and the lower solid line shows the time sequence of departures. The shaded area bounded by the two lines represents the total "work" done by the system in units of job-seconds; let A be the total work. We wish to derive the relationship among L, W, and λ.

a. Figure 2.8b divides the total area into horizontal rectangles, each with a height of one job. Picture sliding all these rectangles to the left so that their left edges line up at $t = 0$. Develop an equation that relates A, C, and W.

b. Figure 2.8c divides the total area into vertical rectangles, defined by the vertical transition boundaries indicated by the dashed lines. Picture sliding all these rectangles down so that their lower edges line up at $N(t) = 0$. Develop an equation that relates A, T, and L.

c. Finally, derive $L = \lambda W$ from the results of (a) and (b).

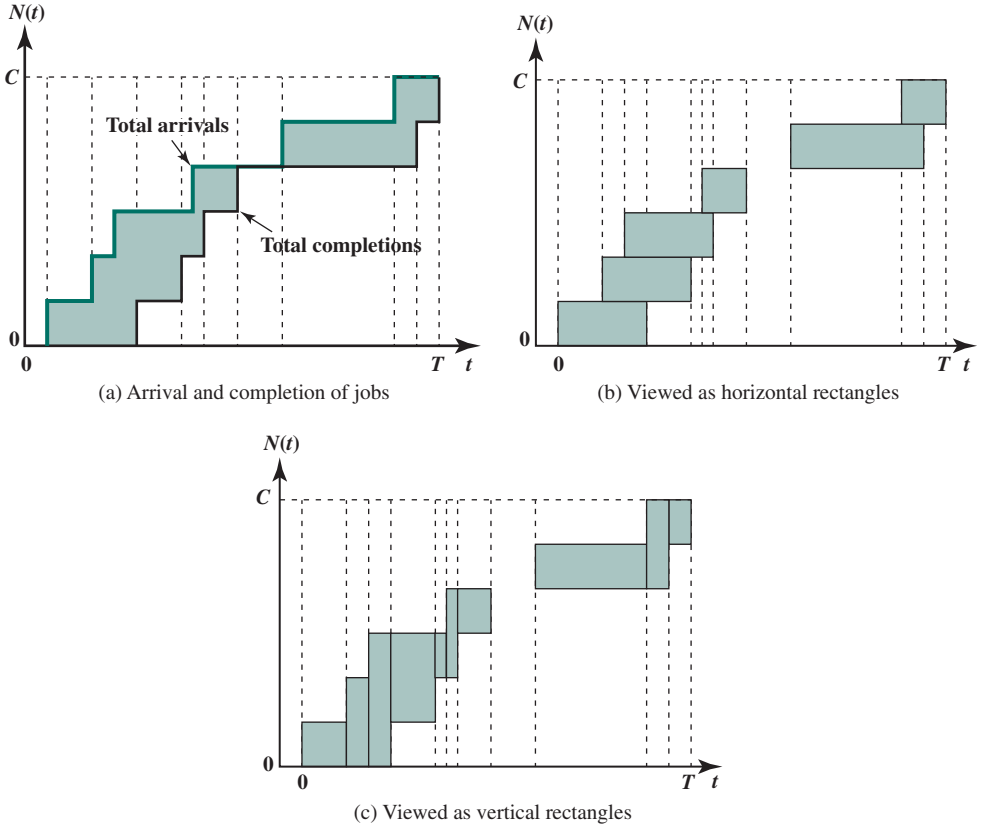
2.11 In Figure 2.8a, jobs arrive at times $t = 0, 1, 1.5, 3.25, 5.25$, and 7.75. The corresponding completion times are $t = 2, 3, 3.5, 4.25, 8.25$, and 8.75.

a. Determine the area of each of the six rectangles in Figure 2.8b and sum to get the total area A. Show your work.

b. Determine the area of each of the 10 rectangles in Figure 2.8c and sum to get the total area A. Show your work.

2.12 In Section 2.6, we specified that the base ratio used for comparing a system under test to a reference system is:

$$r_i = \frac{Tref_i}{Tsut_i}$$



(a) Arrival and completion of jobs

(b) Viewed as horizontal rectangles

(c) Viewed as vertical rectangles

Figure 2.8 Illustration of Little's Law

a. The preceding equation provides a measure of the speedup of the system under test compared to the reference system. Assume that the number of floating-point operations executed in the test program is I_i. Now show the speedup as a function of the instruction execution rate $FLOPS_i$.

b. Another technique for normalizing performance is to express the performance of a system as a percent change relative to the performance of another system. Express this relative change first as a function of instruction execution rate, and then as a function of execution times.

2.13 Assume that a benchmark program executes in 480 seconds on a reference machine A. The same program executes on systems B, C, and D in 360, 540, and 210 seconds, respectively.

a. Show the speedup of each of the three systems under test relative to A.

b. Now show the relative speedup of the three systems. Comment on the three ways of comparing machines (execution time, speedup, relative speedup).

2.14 Repeat the preceding problem using machine D as the reference machine. How does this affect the relative rankings of the four systems?

2.15 Recalculate the results in Table 2.2 using the computer time data of Table 2.4 and comment on the results.

2.16 Equation 2.5 shows two different formulations of the geometric mean, one using a product operator and one using a summation operator.

a. Show that the two formulas are equivalent.

b. Why would the summation formulation be preferred for calculating the geometric mean?

2.17 **Project.** Section 2.5 lists a number of references that document the "benchmark means wars." All of the referenced papers are available at box.com/COA10e. Read these papers and summarize the case for and against the use of the geometric mean for SPEC calculations.

PART TWO THE COMPUTER SYSTEM

CHAPTER 3

A TOP-LEVEL VIEW OF COMPUTER FUNCTION AND INTERCONNECTION

3.1 **Computer Components**

3.2 **Computer Function**
- Instruction Fetch and Execute
- Interrupts
- I/O Function

3.3 **Interconnection Structures**

3.4 **Bus Interconnection**

3.5 **Point-to-Point Interconnect**
- QPI Physical Layer
- QPI Link Layer
- QPI Routing Layer
- QPI Protocol Layer

3.6 **PCI Express**
- PCI Physical and Logical Architecture
- PCIe Physical Layer
- PCIe Transaction Layer
- PCIe Data Link Layer

3.7 **Key Terms, Review Questions, and Problems**

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Understand the basic elements of an instruction cycle and the role of interrupts.
- Describe the concept of interconnection within a computer system.
- Assess the relative advantages of point-to-point interconnection compared to bus interconnection.
- Present an overview of QPI.
- Present an overview of PCIe.

At a top level, a computer consists of CPU (central processing unit), memory, and I/O components, with one or more modules of each type. These components are interconnected in some fashion to achieve the basic function of the computer, which is to execute programs. Thus, at a top level, we can characterize a computer system by describing (1) the external behavior of each component, that is, the data and control signals that it exchanges with other components, and (2) the interconnection structure and the controls required to manage the use of the interconnection structure.

This top-level view of structure and function is important because of its explanatory power in understanding the nature of a computer. Equally important is its use to understand the increasingly complex issues of performance evaluation. A grasp of the top-level structure and function offers insight into system bottlenecks, alternate pathways, the magnitude of system failures if a component fails, and the ease of adding performance enhancements. In many cases, requirements for greater system power and fail-safe capabilities are being met by changing the design rather than merely increasing the speed and reliability of individual components.

This chapter focuses on the basic structures used for computer component interconnection. As background, the chapter begins with a brief examination of the basic components and their interface requirements. Then a functional overview is provided. We are then prepared to examine the use of buses to interconnect system components.

3.1 COMPUTER COMPONENTS

As discussed in Chapter 1, virtually all contemporary computer designs are based on concepts developed by John von Neumann at the Institute for Advanced Studies, Princeton. Such a design is referred to as the *von Neumann architecture* and is based on three key concepts:

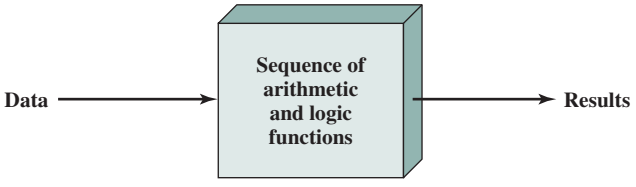
- Data and instructions are stored in a single read–write memory.
- The contents of this memory are addressable by location, without regard to the type of data contained there.

- Execution occurs in a sequential fashion (unless explicitly modified) from one instruction to the next.

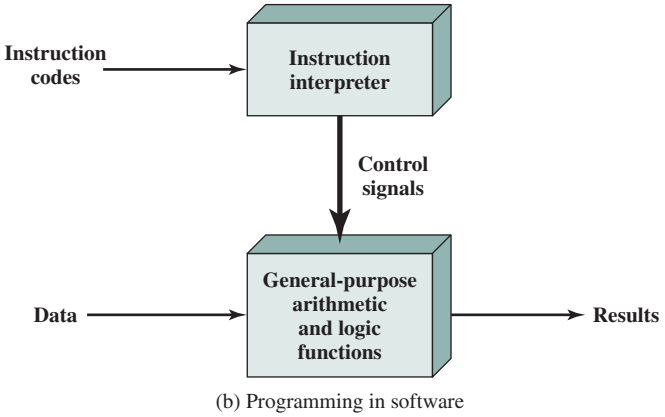
The reasoning behind these concepts was discussed in Chapter 2 but is worth summarizing here. There is a small set of basic logic components that can be combined in various ways to store binary data and perform arithmetic and logical operations on that data. If there is a particular computation to be performed, a configuration of logic components designed specifically for that computation could be constructed. We can think of the process of connecting the various components in the desired configuration as a form of programming. The resulting "program" is in the form of hardware and is termed a *hardwired program*.

Now consider this alternative. Suppose we construct a general-purpose configuration of arithmetic and logic functions. This set of hardware will perform various functions on data depending on control signals applied to the hardware. In the original case of customized hardware, the system accepts data and produces results (Figure 3.1a). With general-purpose hardware, the system accepts data and control signals and produces results. Thus, instead of rewiring the hardware for each new program, the programmer merely needs to supply a new set of control signals.

How shall control signals be supplied? The answer is simple but subtle. The entire program is actually a sequence of steps. At each step, some arithmetic or logical operation is performed on some data. For each step, a new set of control signals is needed. Let us provide a unique code for each possible set of control signals,



(a) Programming in hardware



(b) Programming in software

Figure 3.1 Hardware and Software Approaches

and let us add to the general-purpose hardware a segment that can accept a code and generate control signals (Figure 3.1b).

Programming is now much easier. Instead of rewiring the hardware for each new program, all we need to do is provide a new sequence of codes. Each code is, in effect, an instruction, and part of the hardware interprets each instruction and generates control signals. To distinguish this new method of programming, a sequence of codes or instructions is called *software*.

Figure 3.1b indicates two major components of the system: an instruction interpreter and a module of general-purpose arithmetic and logic functions. These two constitute the CPU. Several other components are needed to yield a functioning computer. Data and instructions must be put into the system. For this we need some sort of input module. This module contains basic components for accepting data and instructions in some form and converting them into an internal form of signals usable by the system. A means of reporting results is needed, and this is in the form of an output module. Taken together, these are referred to as *I/O components*.

One more component is needed. An input device will bring instructions and data in sequentially. But a program is not invariably executed sequentially; it may jump around (e.g., the IAS jump instruction). Similarly, operations on data may require access to more than just one element at a time in a predetermined sequence. Thus, there must be a place to temporarily store both instructions and data. That module is called *memory*, or *main memory*, to distinguish it from external storage or peripheral devices. Von Neumann pointed out that the same memory could be used to store both instructions and data.

Figure 3.2 illustrates these top-level components and suggests the interactions among them. The CPU exchanges data with memory. For this purpose, it typically makes use of two internal (to the CPU) registers: a **memory address register (MAR)**, which specifies the address in memory for the next read or write, and a **memory buffer register (MBR)**, which contains the data to be written into memory or receives the data read from memory. Similarly, an I/O address register (I/OAR) specifies a particular I/O device. An I/O buffer register (I/OBR) is used for the exchange of data between an I/O module and the CPU.

A memory module consists of a set of locations, defined by sequentially numbered addresses. Each location contains a binary number that can be interpreted as either an instruction or data. An I/O module transfers data from external devices to CPU and memory, and vice versa. It contains internal buffers for temporarily holding these data until they can be sent on.

Having looked briefly at these major components, we now turn to an overview of how these components function together to execute programs.

3.2 COMPUTER FUNCTION

The basic function performed by a computer is execution of a program, which consists of a set of instructions stored in memory. The processor does the actual work by executing instructions specified in the program. This section provides an overview of

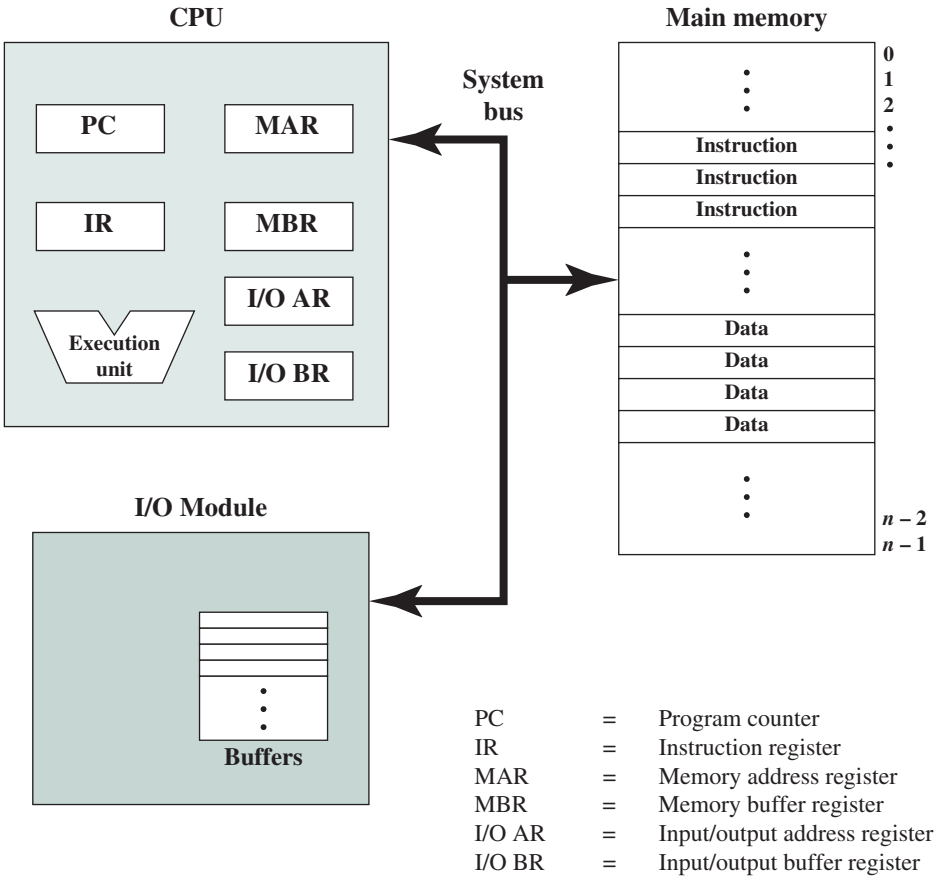


Figure 3.2 Computer Components: Top-Level View

the key elements of program execution. In its simplest form, instruction processing consists of two steps: The processor reads (*fetches*) instructions from memory one at a time and executes each instruction. Program execution consists of repeating the process of instruction fetch and instruction execution. The instruction execution may involve several operations and depends on the nature of the instruction (see, for example, the lower portion of Figure 2.4).

The processing required for a single instruction is called an **instruction cycle**. Using the simplified two-step description given previously, the instruction cycle is depicted in Figure 3.3. The two steps are referred to as the **fetch cycle** and the **execute cycle**. Program execution halts only if the machine is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the computer is encountered.

Instruction Fetch and Execute

At the beginning of each instruction cycle, the processor fetches an instruction from memory. In a typical processor, a register called the program counter (PC) holds the address of the instruction to be fetched next. Unless told otherwise, the processor

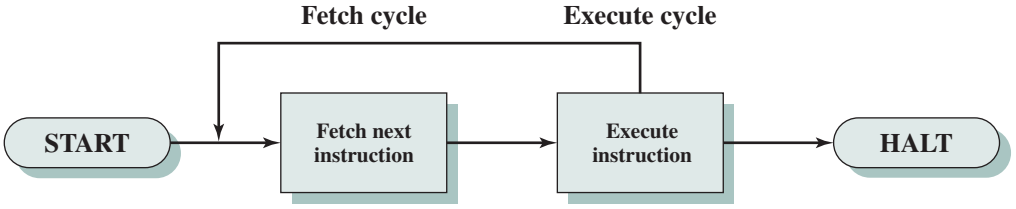


Figure 3.3 Basic Instruction Cycle

always increments the PC after each instruction fetch so that it will fetch the next instruction in sequence (i.e., the instruction located at the next higher memory address). So, for example, consider a computer in which each instruction occupies one 16-bit word of memory. Assume that the program counter is set to memory location 300, where the location address refers to a 16-bit word. The processor will next fetch the instruction at location 300. On succeeding instruction cycles, it will fetch instructions from locations 301, 302, 303, and so on. This sequence may be altered, as explained presently.

The fetched instruction is loaded into a register in the processor known as the instruction register (IR). The instruction contains bits that specify the action the processor is to take. The processor interprets the instruction and performs the required action. In general, these actions fall into four categories:

- **Processor-memory:** Data may be transferred from processor to memory or from memory to processor.
- **Processor-I/O:** Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.
- **Data processing:** The processor may perform some arithmetic or logic operation on data.
- **Control:** An instruction may specify that the sequence of execution be altered. For example, the processor may fetch an instruction from location 149, which specifies that the next instruction be from location 182. The processor will remember this fact by setting the program counter to 182. Thus, on the next fetch cycle, the instruction will be fetched from location 182 rather than 150.

An instruction's execution may involve a combination of these actions.

Consider a simple example using a hypothetical machine that includes the characteristics listed in Figure 3.4. The processor contains a single data register, called an accumulator (AC). Both instructions and data are 16 bits long. Thus, it is convenient to organize memory using 16-bit words. The instruction format provides 4 bits for the opcode, so that there can be as many as $2^4 = 16$ different opcodes, and up to $2^{12} = 4096$ (4K) words of memory can be directly addressed.

Figure 3.5 illustrates a partial program execution, showing the relevant portions of memory and processor registers.[1] The program fragment shown adds the contents of the memory word at address 940 to the contents of the memory word at

[1]Hexadecimal notation is used, in which each digit represents 4 bits. This is the most convenient notation for representing the contents of memory and registers when the word length is a multiple of 4. See Chapter 9 for a basic refresher on number systems (decimal, binary, hexadecimal).

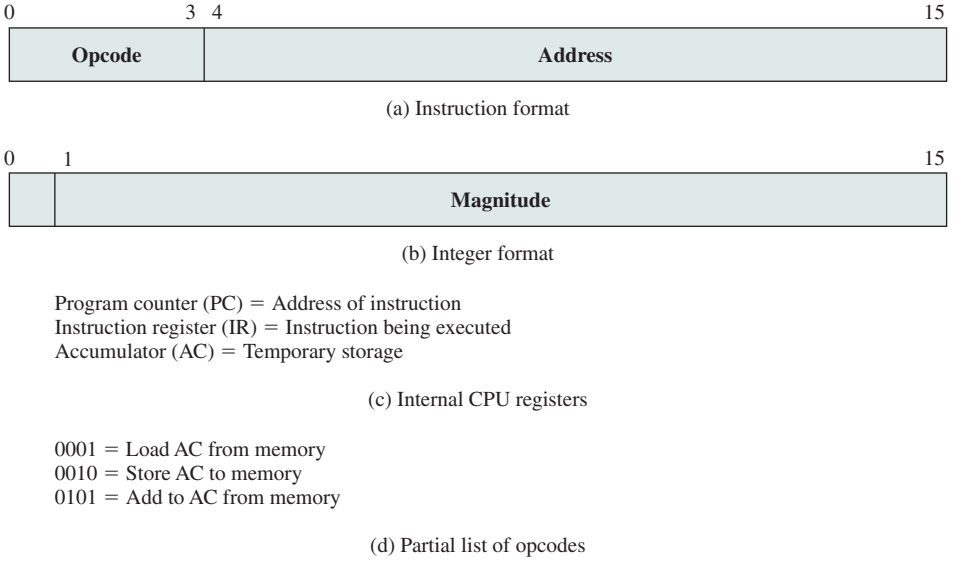


Figure 3.4 Characteristics of a Hypothetical Machine

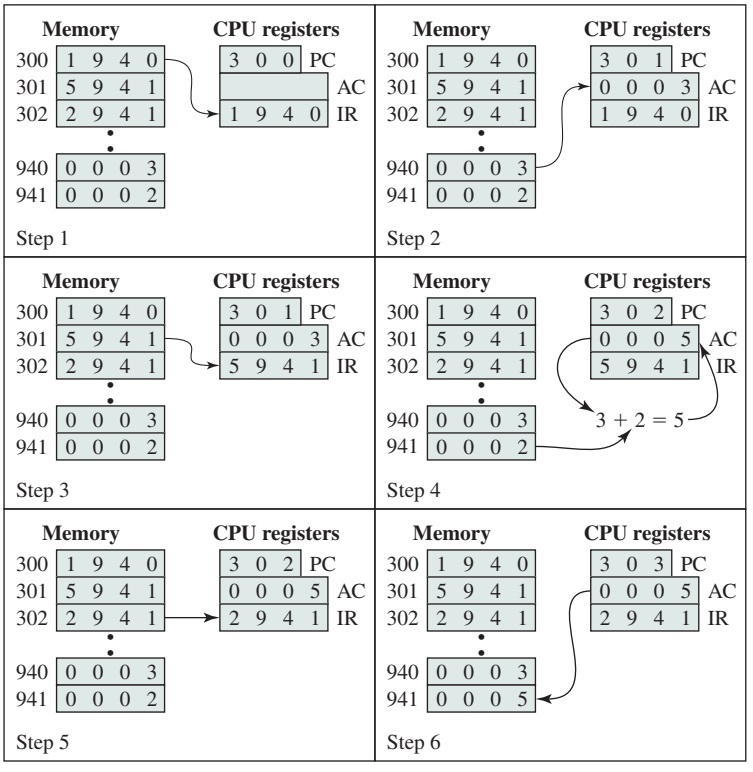


Figure 3.5 Example of Program Execution (contents of memory and registers in hexadecimal)

address 941 and stores the result in the latter location. Three instructions, which can be described as three fetch and three execute cycles, are required:

1. The PC contains 300, the address of the first instruction. This instruction (the value 1940 in hexadecimal) is loaded into the instruction register IR, and the PC is incremented. Note that this process involves the use of a memory address register and a memory buffer register. For simplicity, these intermediate registers are ignored.
2. The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded. The remaining 12 bits (three hexadecimal digits) specify the address (940) from which data are to be loaded.
3. The next instruction (5941) is fetched from location 301, and the PC is incremented.
4. The old contents of the AC and the contents of location 941 are added, and the result is stored in the AC.
5. The next instruction (2941) is fetched from location 302, and the PC is incremented.
6. The contents of the AC are stored in location 941.

In this example, three instruction cycles, each consisting of a fetch cycle and an execute cycle, are needed to add the contents of location 940 to the contents of 941. With a more complex set of instructions, fewer cycles would be needed. Some older processors, for example, included instructions that contain more than one memory address. Thus, the execution cycle for a particular instruction on such processors could involve more than one reference to memory. Also, instead of memory references, an instruction may specify an I/O operation.

For example, the PDP-11 processor includes an instruction, expressed symbolically as ADD B,A, that stores the sum of the contents of memory locations B and A into memory location A. A single instruction cycle with the following steps occurs:

- Fetch the ADD instruction.
- Read the contents of memory location A into the processor.
- Read the contents of memory location B into the processor. In order that the contents of A are not lost, the processor must have at least two registers for storing memory values, rather than a single accumulator.
- Add the two values.
- Write the result from the processor to memory location A.

Thus, the execution cycle for a particular instruction may involve more than one reference to memory. Also, instead of memory references, an instruction may specify an I/O operation. With these additional considerations in mind, Figure 3.6 provides a more detailed look at the basic instruction cycle of Figure 3.3. The figure is in the form of a state diagram. For any given instruction cycle, some states may be null and others may be visited more than once. The states can be described as follows:

- **Instruction address calculation (iac):** Determine the address of the next instruction to be executed. Usually, this involves adding a fixed number to

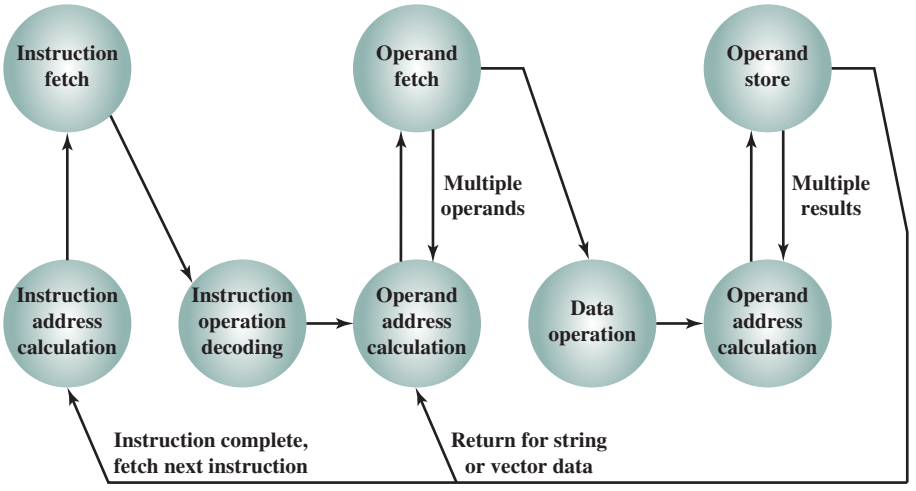


Figure 3.6 Instruction Cycle State Diagram

the address of the previous instruction. For example, if each instruction is 16 bits long and memory is organized into 16-bit words, then add 1 to the previous address. If, instead, memory is organized as individually addressable 8-bit bytes, then add 2 to the previous address.

- **Instruction fetch (if):** Read instruction from its memory location into the processor.
- **Instruction operation decoding (iod):** Analyze instruction to determine type of operation to be performed and operand(s) to be used.
- **Operand address calculation (oac):** If the operation involves reference to an operand in memory or available via I/O, then determine the address of the operand.
- **Operand fetch (of):** Fetch the operand from memory or read it in from I/O.
- **Data operation (do):** Perform the operation indicated in the instruction.
- **Operand store (os):** Write the result into memory or out to I/O.

States in the upper part of Figure 3.6 involve an exchange between the processor and either memory or an I/O module. States in the lower part of the diagram involve only internal processor operations. The oac state appears twice, because an instruction may involve a read, a write, or both. However, the action performed during that state is fundamentally the same in both cases, and so only a single state identifier is needed.

Also note that the diagram allows for multiple operands and multiple results, because some instructions on some machines require this. For example, the PDP-11 instruction ADD A,B results in the following sequence of states: iac, if, iod, oac, of, oac, of, do, oac, os.

Finally, on some machines, a single instruction can specify an operation to be performed on a vector (one-dimensional array) of numbers or a string (one-dimensional

array) of characters. As Figure 3.6 indicates, this would involve repetitive operand fetch and/or store operations.

Interrupts

Virtually all computers provide a mechanism by which other modules (I/O, memory) may **interrupt** the normal processing of the processor. Table 3.1 lists the most common classes of interrupts. The specific nature of these interrupts is examined later in this book, especially in Chapters 7 and 14. However, we need to introduce the concept now to understand more clearly the nature of the instruction cycle and the implications of interrupts on the interconnection structure. The reader need not be concerned at this stage about the details of the generation and processing of interrupts, but only focus on the communication between modules that results from interrupts.

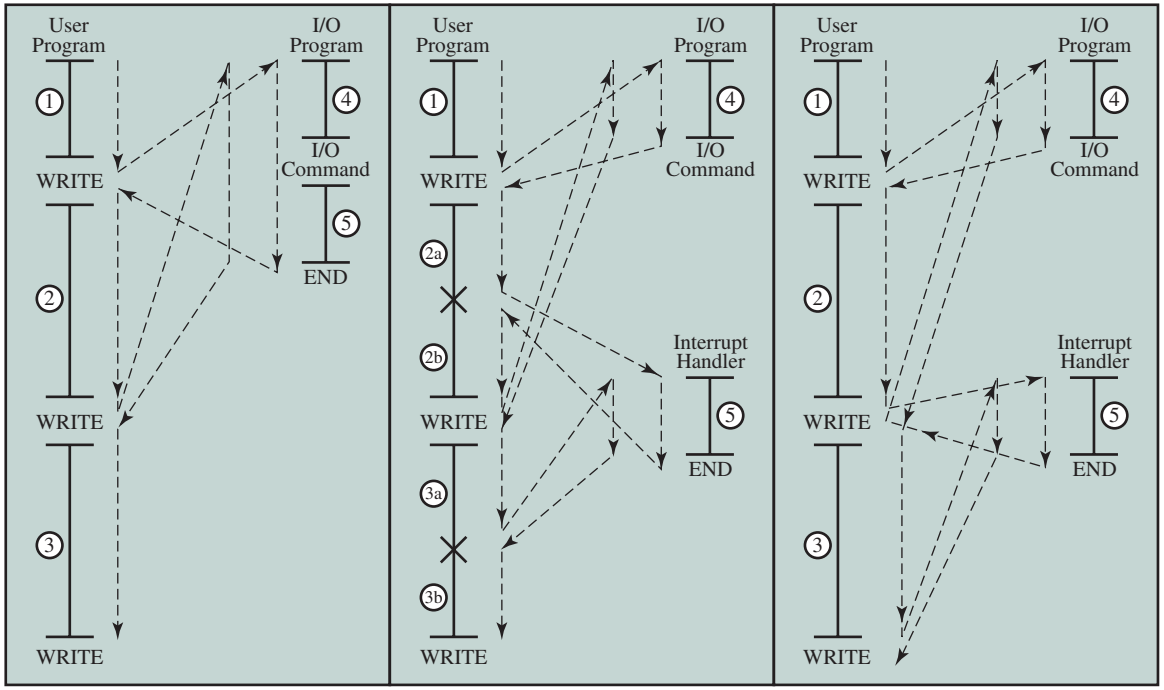
Interrupts are provided primarily as a way to improve processing efficiency. For example, most external devices are much slower than the processor. Suppose that the processor is transferring data to a printer using the instruction cycle scheme of Figure 3.3. After each write operation, the processor must pause and remain idle until the printer catches up. The length of this pause may be on the order of many hundreds or even thousands of instruction cycles that do not involve memory. Clearly, this is a very wasteful use of the processor.

Figure 3.7a illustrates this state of affairs. The user program performs a series of WRITE calls interleaved with processing. Code segments 1, 2, and 3 refer to sequences of instructions that do not involve I/O. The WRITE calls are to an I/O program that is a system utility and that will perform the actual I/O operation. The I/O program consists of three sections:

- A sequence of instructions, labeled 4 in the figure, to prepare for the actual I/O operation. This may include copying the data to be output into a special buffer and preparing the parameters for a device command.
- The actual I/O command. Without the use of interrupts, once this command is issued, the program must wait for the I/O device to perform the requested function (or periodically poll the device). The program might wait by simply repeatedly performing a test operation to determine if the I/O operation is done.
- A sequence of instructions, labeled 5 in the figure, to complete the operation. This may include setting a flag indicating the success or failure of the operation.

Table 3.1 Classes of Interrupts

Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, or reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation, request service from the processor, or to signal a variety of error conditions.
Hardware Failure	Generated by a failure such as power failure or memory parity error.



(a) No interrupts (b) Interrupts; short I/O wait (c) Interrupts; long I/O wait

× = interrupt occurs during course of execution of user program

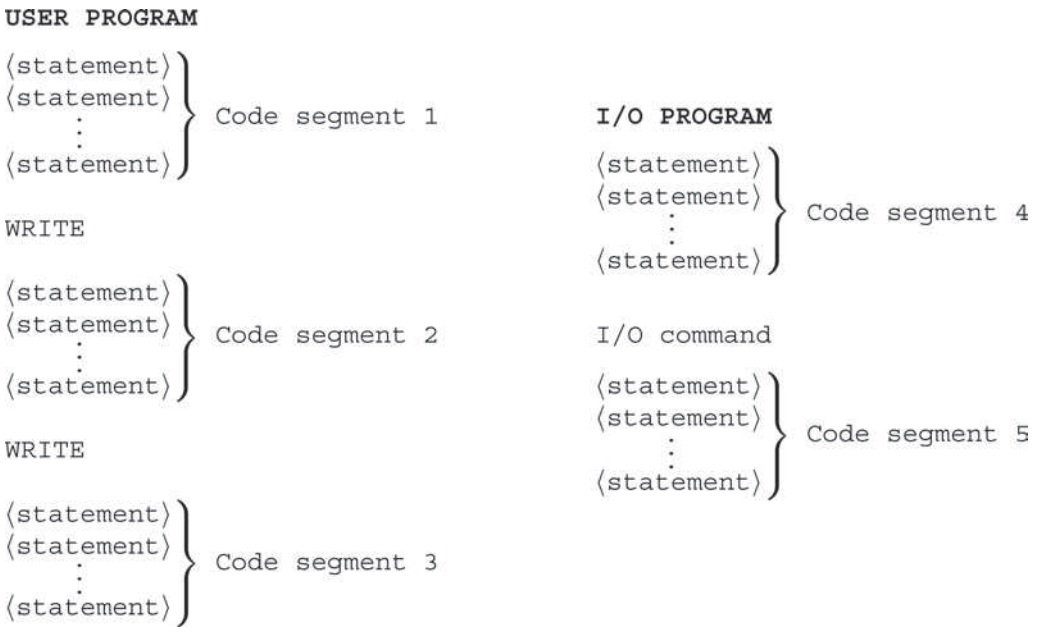
Figure 3.7 Program Flow of Control without and with Interrupts

Because the I/O operation may take a relatively long time to complete, the I/O program is hung up waiting for the operation to complete; hence, the user program is stopped at the point of the WRITE call for some considerable period of time.

INTERRUPTS AND THE INSTRUCTION CYCLE With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress. Consider the flow of control in Figure 3.7b. As before, the user program reaches a point at which it makes a system call in the form of a WRITE call. The I/O program that is invoked in this case consists only of the preparation code and the actual I/O command. After these few instructions have been executed, control returns to the user program. Meanwhile, the external device is busy accepting data from computer memory and printing it. This I/O operation is conducted concurrently with the execution of instructions in the user program.

When the external device becomes ready to be serviced—that is, when it is ready to accept more data from the processor—the I/O module for that external device sends an *interrupt request* signal to the processor. The processor responds by suspending operation of the current program, branching off to a program to service that particular I/O device, known as an **interrupt handler**, and resuming the original execution after the device is serviced. The points at which such interrupts occur are indicated by an asterisk in Figure 3.7b.

Let us try to clarify what is happening in Figure 3.7. We have a user program that contains two WRITE commands. There is a segment of code at the beginning, then one WRITE command, then a second segment of code, then a second WRITE command, then a third and final segment of code. The WRITE command invokes the I/O program provided by the OS. Similarly, the I/O program consists of a segment of code, followed by an I/O command, followed by another segment of code. The I/O command invokes a hardware I/O operation.



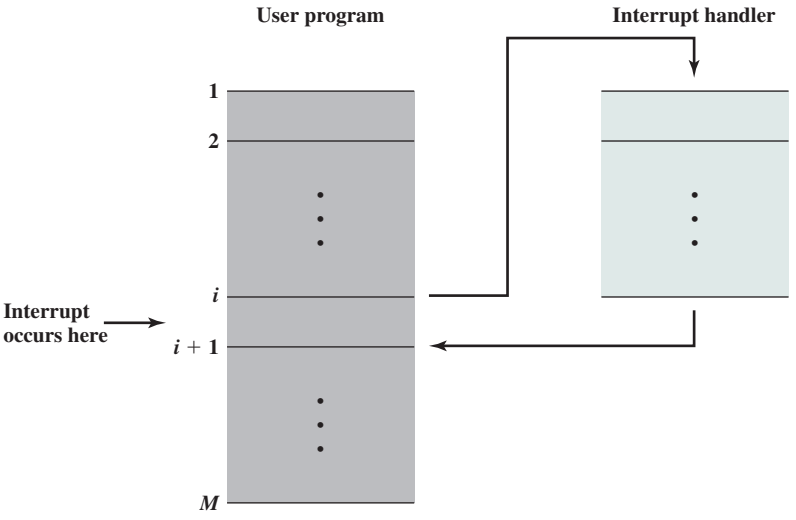


Figure 3.8 Transfer of Control via Interrupts

From the point of view of the user program, an interrupt is just that: an interruption of the normal sequence of execution. When the interrupt processing is completed, execution resumes (Figure 3.8). Thus, the user program does not have to contain any special code to accommodate interrupts; the processor and the operating system are responsible for suspending the user program and then resuming it at the same point.

To accommodate interrupts, an *interrupt cycle* is added to the instruction cycle, as shown in Figure 3.9. In the interrupt cycle, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal. If no interrupts are pending, the processor proceeds to the fetch cycle and fetches the next instruction of the current program. If an interrupt is pending, the processor does the following:

- It suspends execution of the current program being executed and saves its context. This means saving the address of the next instruction to be executed

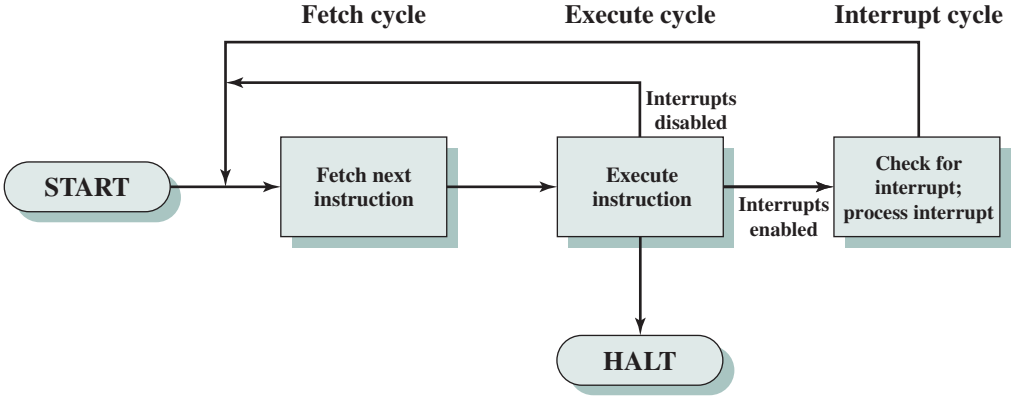


Figure 3.9 Instruction Cycle with Interrupts

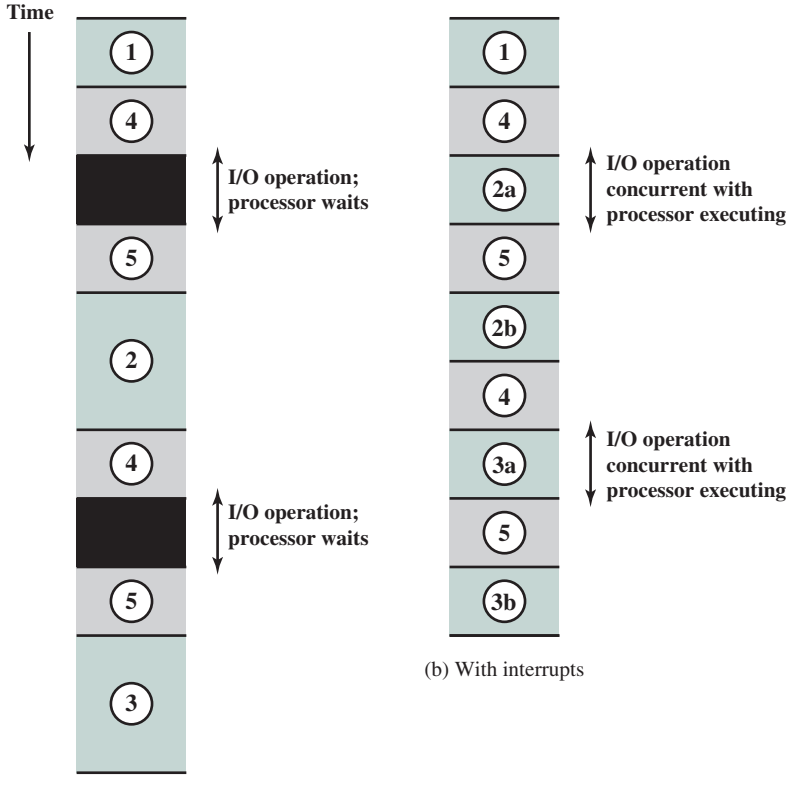
(current contents of the program counter) and any other data relevant to the processor's current activity.

- It sets the program counter to the starting address of an *interrupt handler* routine.

The processor now proceeds to the fetch cycle and fetches the first instruction in the interrupt handler program, which will service the interrupt. The interrupt handler program is generally part of the operating system. Typically, this program determines the nature of the interrupt and performs whatever actions are needed. In the example we have been using, the handler determines which I/O module generated the interrupt and may branch to a program that will write more data out to that I/O module. When the interrupt handler routine is completed, the processor can resume execution of the user program at the point of interruption.

It is clear that there is some overhead involved in this process. Extra instructions must be executed (in the interrupt handler) to determine the nature of the interrupt and to decide on the appropriate action. Nevertheless, because of the relatively large amount of time that would be wasted by simply waiting on an I/O operation, the processor can be employed much more efficiently with the use of interrupts.

To appreciate the gain in efficiency, consider Figure 3.10, which is a timing diagram based on the flow of control in Figures 3.7a and 3.7b. In this figure, user program code segments are shaded green, and I/O program code segments are



(a) Without interrupts

(b) With interrupts

Figure 3.10 Program Timing: Short I/O Wait

shaded gray. Figure 3.10a shows the case in which interrupts are not used. The processor must wait while an I/O operation is performed.

Figures 3.7b and 3.10b assume that the time required for the I/O operation is relatively short: less than the time to complete the execution of instructions between write operations in the user program. In this case, the segment of code labeled code segment 2 is interrupted. A portion of the code (2a) executes (while the I/O operation is performed) and then the interrupt occurs (upon the completion of the I/O operation). After the interrupt is serviced, execution resumes with the remainder of code segment 2 (2b).

The more typical case, especially for a slow device such as a printer, is that the I/O operation will take much more time than executing a sequence of user instructions. Figure 3.7c indicates this state of affairs. In this case, the user program reaches the second WRITE call before the I/O operation spawned by the first call is complete. The result is that the user program is hung up at that point. When the preceding I/O operation is completed, this new WRITE call may be processed, and a new I/O operation may be started. Figure 3.11 shows the timing for this situation with

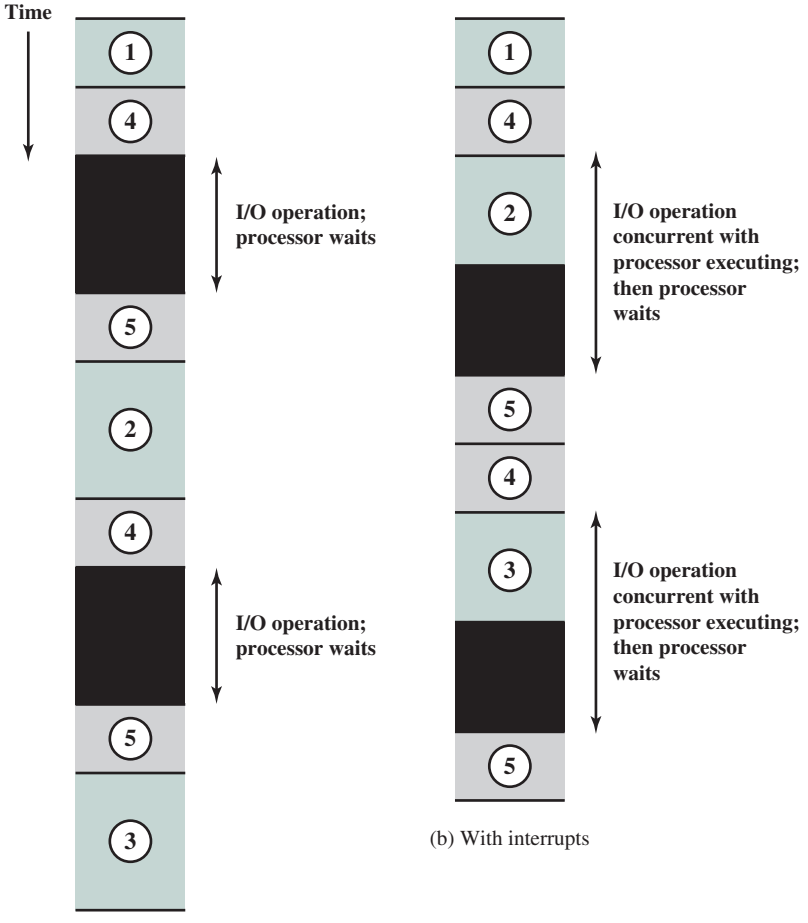


Figure 3.11 Program Timing: Long I/O Wait

and without the use of interrupts. We can see that there is still a gain in efficiency because part of the time during which the I/O operation is under way overlaps with the execution of user instructions.

Figure 3.12 shows a revised instruction cycle state diagram that includes interrupt cycle processing.

MULTIPLE INTERRUPTS The discussion so far has focused only on the occurrence of a single interrupt. Suppose, however, that multiple interrupts can occur. For example, a program may be receiving data from a communications line and printing results. The printer will generate an interrupt every time it completes a print operation. The communication line controller will generate an interrupt every time a unit of data arrives. The unit could either be a single character or a block, depending on the nature of the communications discipline. In any case, it is possible for a communications interrupt to occur while a printer interrupt is being processed.

Two approaches can be taken to dealing with multiple interrupts. The first is to disable interrupts while an interrupt is being processed. A **disabled interrupt** simply means that the processor can and will ignore that interrupt request signal. If an interrupt occurs during this time, it generally remains pending and will be checked by the processor after the processor has enabled interrupts. Thus, when a user program is executing and an interrupt occurs, interrupts are disabled immediately. After the interrupt handler routine completes, interrupts are enabled before resuming the user program, and the processor checks to see if additional interrupts have occurred. This approach is nice and simple, as interrupts are handled in strict sequential order (Figure 3.13a).

The drawback to the preceding approach is that it does not take into account relative priority or time-critical needs. For example, when input arrives from the communications line, it may need to be absorbed rapidly to make room for more input. If the first batch of input has not been processed before the second batch arrives, data may be lost.

A second approach is to define priorities for interrupts and to allow an interrupt of higher priority to cause a lower-priority interrupt handler to be itself interrupted (Figure 3.13b). As an example of this second approach, consider a system with three I/O devices: a printer, a disk, and a communications line, with increasing priorities of 2, 4, and 5, respectively. Figure 3.14 illustrates a possible sequence. A user program begins at $t = 0$. At $t = 10$, a printer interrupt occurs; user information is placed on the system stack and execution continues at the printer **interrupt service routine (ISR)**. While this routine is still executing, at $t = 15$, a communications interrupt occurs. Because the communications line has higher priority than the printer, the interrupt is honored. The printer ISR is interrupted, its state is pushed onto the stack, and execution continues at the communications ISR. While this routine is executing, a disk interrupt occurs ($t = 20$). Because this interrupt is of lower priority, it is simply held, and the communications ISR runs to completion.

When the communications ISR is complete ($t = 25$), the previous processor state is restored, which is the execution of the printer ISR. However, before even a single instruction in that routine can be executed, the processor honors the higher-priority disk interrupt and control transfers to the disk ISR. Only when that

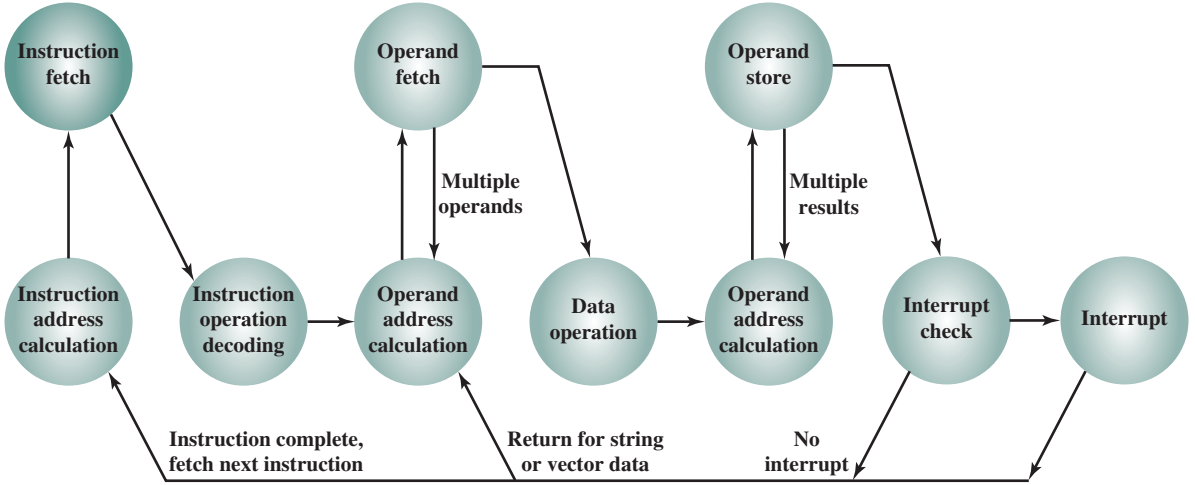
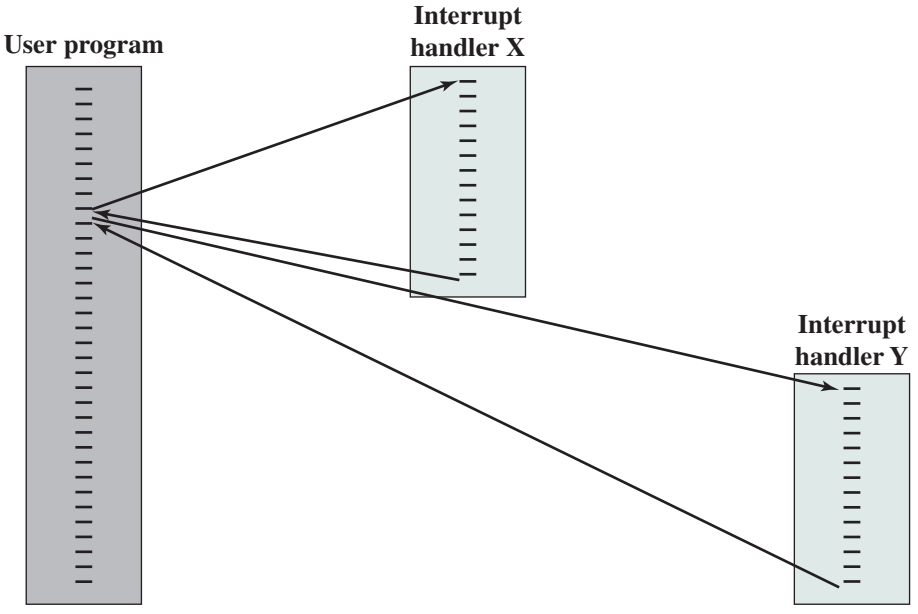
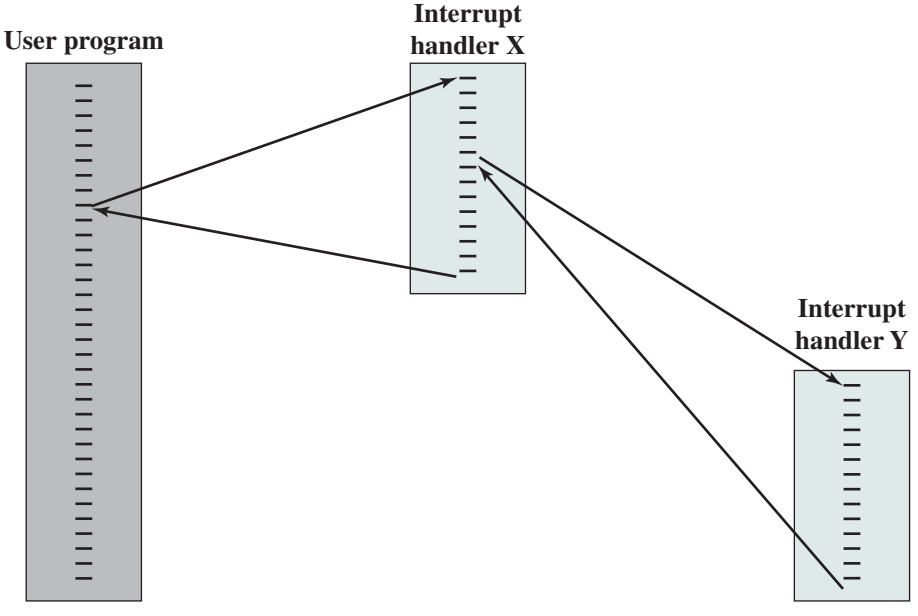


Figure 3.12 Instruction Cycle State Diagram, with Interrupts



(a) Sequential interrupt processing



(b) Nested interrupt processing

Figure 3.13 Transfer of Control with Multiple Interrupts

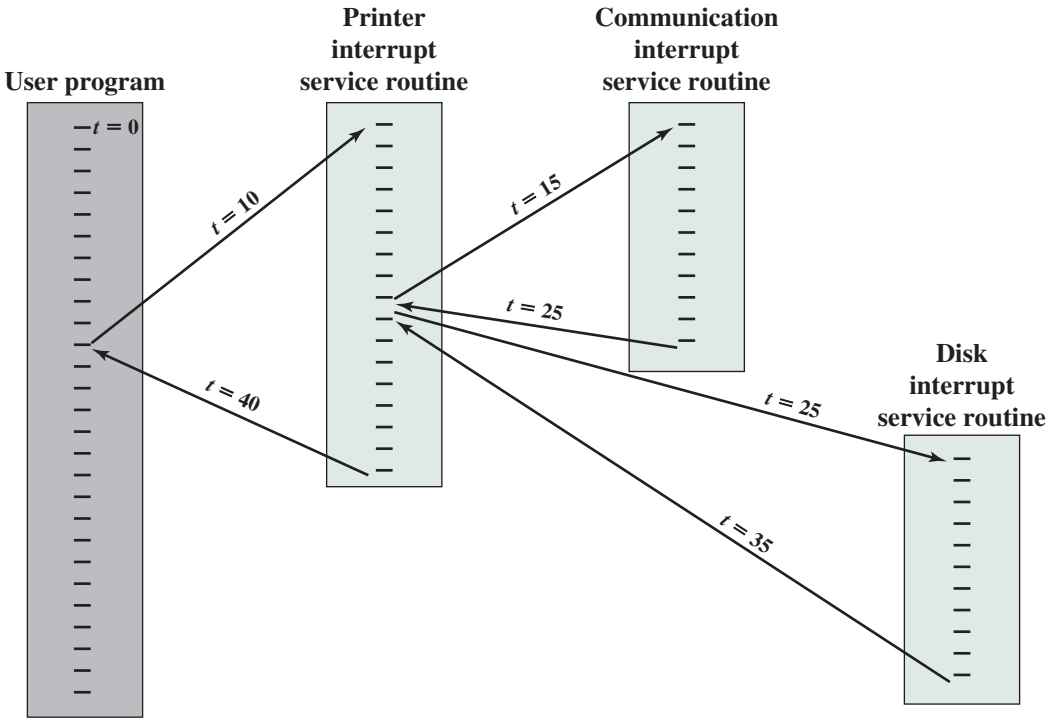


Figure 3.14 Example Time Sequence of Multiple Interrupts

routine is complete ($t = 35$) is the printer ISR resumed. When that routine completes ($t = 40$), control finally returns to the user program.

I/O Function

Thus far, we have discussed the operation of the computer as controlled by the processor, and we have looked primarily at the interaction of processor and memory. The discussion has only alluded to the role of the I/O component. This role is discussed in detail in Chapter 7, but a brief summary is in order here.

An I/O module (e.g., a disk controller) can exchange data directly with the processor. Just as the processor can initiate a read or write with memory, designating the address of a specific location, the processor can also read data from or write data to an I/O module. In this latter case, the processor identifies a specific device that is controlled by a particular I/O module. Thus, an instruction sequence similar in form to that of Figure 3.5 could occur, with I/O instructions rather than memory-referencing instructions.

In some cases, it is desirable to allow I/O exchanges to occur directly with memory. In such a case, the processor grants to an I/O module the authority to read from or write to memory, so that the I/O-memory transfer can occur without tying up the processor. During such a transfer, the I/O module issues read or write commands to memory, relieving the processor of responsibility for the exchange. This operation is known as direct memory access (DMA) and is examined in Chapter 7.

3.3 INTERCONNECTION STRUCTURES

A computer consists of a set of components or modules of three basic types (processor, memory, I/O) that communicate with each other. In effect, a computer is a network of basic modules. Thus, there must be paths for connecting the modules.

The collection of paths connecting the various modules is called the *interconnection structure*. The design of this structure will depend on the exchanges that must be made among modules.

Figure 3.15 suggests the types of exchanges that are needed by indicating the major forms of input and output for each module type[2]:

- **Memory:** Typically, a memory module will consist of N words of equal length. Each word is assigned a unique numerical address (0, 1, . . . , $N-1$). A word of data can be read from or written into the memory. The nature of the operation

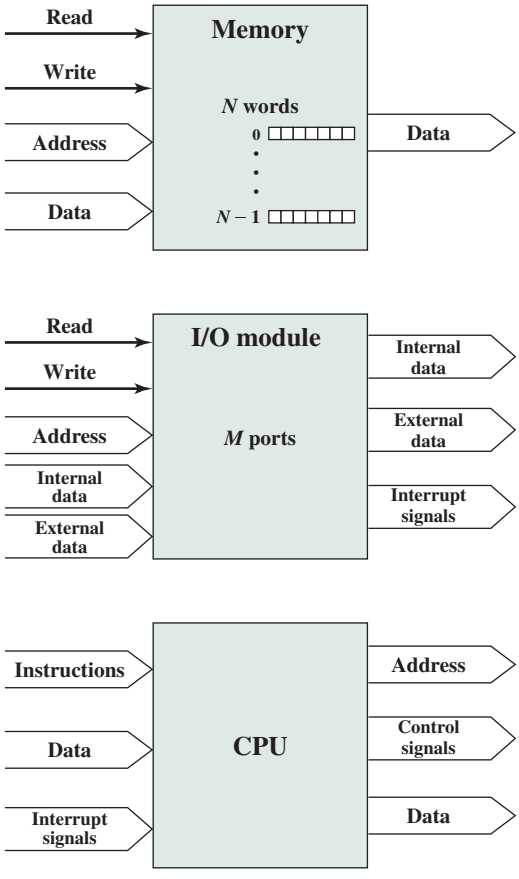


Figure 3.15 Computer Modules

[2]The wide arrows represent multiple signal lines carrying multiple bits of information in parallel. Each narrow arrow represents a single signal line.

is indicated by read and write control signals. The location for the operation is specified by an address.

- **I/O module:** From an internal (to the computer system) point of view, I/O is functionally similar to memory. There are two operations; read and write. Further, an I/O module may control more than one external device. We can refer to each of the interfaces to an external device as a *port* and give each a unique address (e.g., 0, 1, . . . , $M-1$). In addition, there are external data paths for the input and output of data with an external device. Finally, an I/O module may be able to send interrupt signals to the processor.
- **Processor:** The processor reads in instructions and data, writes out data after processing, and uses control signals to control the overall operation of the system. It also receives interrupt signals.

The preceding list defines the data to be exchanged. The interconnection structure must support the following types of transfers:

- **Memory to processor:** The processor reads an instruction or a unit of data from memory.
- **Processor to memory:** The processor writes a unit of data to memory.
- **I/O to processor:** The processor reads data from an I/O device via an I/O module.
- **Processor to I/O:** The processor sends data to the I/O device.
- **I/O to or from memory:** For these two cases, an I/O module is allowed to exchange data directly with memory, without going through the processor, using direct memory access.

Over the years, a number of interconnection structures have been tried. By far the most common are (1) the **bus** and various multiple-bus structures, and (2) point-to-point interconnection structures with packetized data transfer. We devote the remainder of this chapter for a discussion of these structures.

3.4 BUS INTERCONNECTION

The bus was the dominant means of computer system component interconnection for decades. For general-purpose computers, it has gradually given way to various point-to-point interconnection structures, which now dominate computer system design. However, bus structures are still commonly used for embedded systems, particularly microcontrollers. In this section, we give a brief overview of bus structure. Appendix C provides more detail.

A bus is a communication pathway connecting two or more devices. A key characteristic of a bus is that it is a shared transmission medium. Multiple devices connect to the bus, and a signal transmitted by any one device is available for reception by all other devices attached to the bus. If two devices transmit during the same time period, their signals will overlap and become garbled. Thus, only one device at a time can successfully transmit.

Typically, a bus consists of multiple communication pathways, or lines. Each line is capable of transmitting signals representing binary 1 and binary 0. Over time, a sequence of binary digits can be transmitted across a single line. Taken together, several lines of a bus can be used to transmit binary digits simultaneously (in parallel). For example, an 8-bit unit of data can be transmitted over eight bus lines.

Computer systems contain a number of different buses that provide pathways between components at various levels of the computer system hierarchy. A bus that connects major computer components (processor, memory, I/O) is called a **system bus**. The most common computer interconnection structures are based on the use of one or more system buses.

A system bus consists, typically, of from about fifty to hundreds of separate lines. Each line is assigned a particular meaning or function. Although there are many different bus designs, on any bus the lines can be classified into three functional groups (Figure 3.16): data, address, and control lines. In addition, there may be power distribution lines that supply power to the attached modules.

The **data lines** provide a path for moving data among system modules. These lines, collectively, are called the **data bus**. The data bus may consist of 32, 64, 128, or even more separate lines, the number of lines being referred to as the *width* of the data bus. Because each line can carry only one bit at a time, the number of lines determines how many bits can be transferred at a time. The width of the data bus is a key factor in determining overall system performance. For example, if the data bus is 32 bits wide and each instruction is 64 bits long, then the processor must access the memory module twice during each instruction cycle.

The **address lines** are used to designate the source or destination of the data on the data bus. For example, if the processor wishes to read a word (8, 16, or 32 bits) of data from memory, it puts the address of the desired word on the address lines. Clearly, the width of the **address bus** determines the maximum possible memory capacity of the system. Furthermore, the address lines are generally also used to address I/O ports. Typically, the higher-order bits are used to select a particular module on the bus, and the lower-order bits select a memory location or I/O port within the module. For example, on an 8-bit address bus, address 01111111 and below might reference locations in a memory module (module 0) with 128 words of memory, and address 10000000 and above refer to devices attached to an I/O module (module 1).

The **control lines** are used to control the access to and the use of the data and address lines. Because the data and address lines are shared by all components,

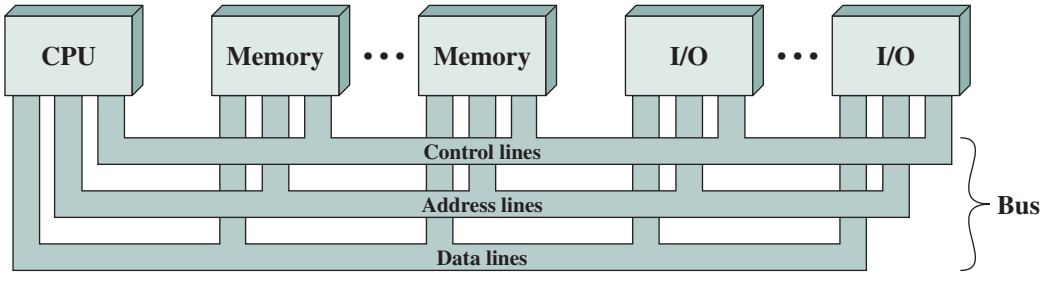


Figure 3.16 Bus Interconnection Scheme

there must be a means of controlling their use. Control signals transmit both command and timing information among system modules. Timing signals indicate the validity of data and address information. Command signals specify operations to be performed. Typical control lines include:

- **Memory write:** causes data on the bus to be written into the addressed location.
- **Memory read:** causes data from the addressed location to be placed on the bus.
- **I/O write:** causes data on the bus to be output to the addressed I/O port.
- **I/O read:** causes data from the addressed I/O port to be placed on the bus.
- **Transfer ACK:** indicates that data have been accepted from or placed on the bus.
- **Bus request:** indicates that a module needs to gain control of the bus.
- **Bus grant:** indicates that a requesting module has been granted control of the bus.
- **Interrupt request:** indicates that an interrupt is pending.
- **Interrupt ACK:** acknowledges that the pending interrupt has been recognized.
- **Clock:** is used to synchronize operations.
- **Reset:** initializes all modules.

The operation of the bus is as follows. If one module wishes to send data to another, it must do two things: (1) obtain the use of the bus, and (2) transfer data via the bus. If one module wishes to request data from another module, it must (1) obtain the use of the bus, and (2) transfer a request to the other module over the appropriate control and address lines. It must then wait for that second module to send the data.

3.5 POINT-TO-POINT INTERCONNECT

The shared bus architecture was the standard approach to interconnection between the processor and other components (memory, I/O, and so on) for decades. But contemporary systems increasingly rely on point-to-point interconnection rather than shared buses.

The principal reason driving the change from bus to point-to-point interconnect was the electrical constraints encountered with increasing the frequency of wide synchronous buses. At higher and higher data rates, it becomes increasingly difficult to perform the synchronization and arbitration functions in a timely fashion. Further, with the advent of multicore chips, with multiple processors and significant memory on a single chip, it was found that the use of a conventional shared bus on the same chip magnified the difficulties of increasing bus data rate and reducing bus latency to keep up with the processors. Compared to the shared bus, the point-to-point interconnect has lower latency, higher data rate, and better scalability.

In this section, we look at an important and representative example of the point-to-point interconnect approach: Intel's **QuickPath Interconnect (QPI)**, which was introduced in 2008.

The following are significant characteristics of QPI and other point-to-point interconnect schemes:

- **Multiple direct connections:** Multiple components within the system enjoy direct pairwise connections to other components. This eliminates the need for arbitration found in shared transmission systems.
- **Layered protocol architecture:** As found in network environments, such as TCP/IP-based data networks, these processor-level interconnects use a layered protocol architecture, rather than the simple use of control signals found in shared bus arrangements.
- **Packetized data transfer:** Data are not sent as a raw bit stream. Rather, data are sent as a sequence of packets, each of which includes control headers and error control codes.

Figure 3.17 illustrates a typical use of QPI on a multicore computer. The QPI links (indicated by the green arrow pairs in the figure) form a switching fabric that enables data to move throughout the network. Direct QPI connections can be established between each pair of core processors. If core A in Figure 3.17 needs to access the memory controller in core D, it sends its request through either cores B or C, which must in turn forward that request on to the memory controller in core D. Similarly, larger systems with eight or more processors can be built using processors with three links and routing traffic through intermediate processors.

In addition, QPI is used to connect to an I/O module, called an I/O hub (IOH). The IOH acts as a switch directing traffic to and from I/O devices. Typically in newer

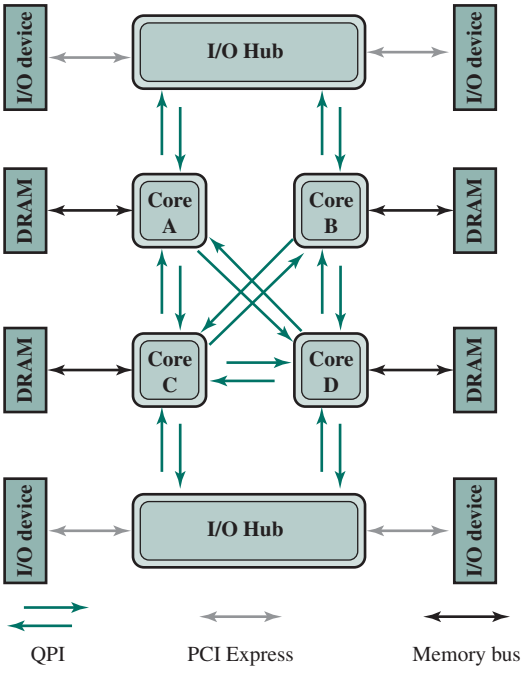


Figure 3.17 Multicore Configuration Using QPI

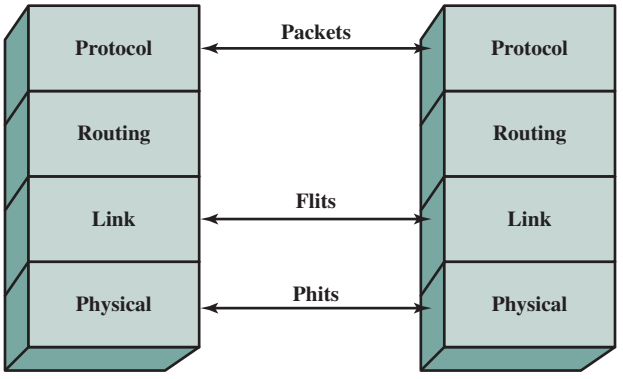


Figure 3.18 QPI Layers

systems, the link from the IOH to the I/O device controller uses an interconnect technology called PCI Express (PCIe), described later in this chapter. The IOH translates between the QPI protocols and formats and the PCIe protocols and formats. A core also links to a main memory module (typically the memory uses dynamic access random memory (DRAM) technology) using a dedicated memory bus.

QPI is defined as a four-layer protocol architecture,[3] encompassing the following layers (Figure 3.18):

- **Physical:** Consists of the actual wires carrying the signals, as well as circuitry and logic to support ancillary features required in the transmission and receipt of the 1s and 0s. The unit of transfer at the Physical layer is 20 bits, which is called a **Phit** (physical unit).
- **Link:** Responsible for reliable transmission and flow control. The Link layer's unit of transfer is an 80-bit **Flit** (flow control unit).
- **Routing:** Provides the framework for directing packets through the fabric.
- **Protocol:** The high-level set of rules for exchanging **packets** of data between devices. A packet is comprised of an integral number of Flits.

QPI Physical Layer

Figure 3.19 shows the physical architecture of a QPI port. The QPI port consists of 84 individual links grouped as follows. Each data path consists of a pair of wires that transmits data one bit at a time; the pair is referred to as a **lane**. There are 20 data lanes in each direction (transmit and receive), plus a clock lane in each direction. Thus, QPI is capable of transmitting 20 bits in parallel in each direction. The 20-bit unit is referred to as a *phit*. Typical signaling speeds of the link in current products calls for operation at 6.4 GT/s (transfers per second). At 20 bits per transfer, that adds up to 16 GB/s, and since QPI links involve dedicated bidirectional pairs, the total capacity is 32 GB/s.

[3] The reader unfamiliar with the concept of a protocol architecture will find a brief overview in Appendix D.

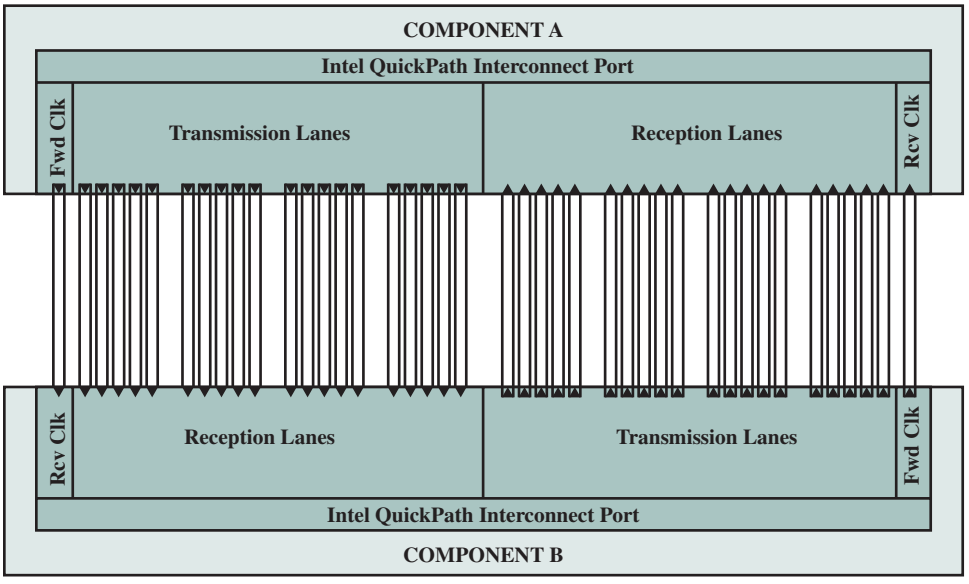


Figure 3.19 Physical Interface of the Intel QPI Interconnect

The lanes in each direction are grouped into four quadrants of 5 lanes each. In some applications, the link can also operate at half or quarter widths in order to reduce power consumption or work around failures.

The form of transmission on each lane is known as **differential signaling**, or **balanced transmission**. With balanced transmission, signals are transmitted as a current that travels down one conductor and returns on the other. The binary value depends on the voltage difference. Typically, one line has a positive voltage value and the other line has zero voltage, and one line is associated with binary 1 and one line is associated with binary 0. Specifically, the technique used by QPI is known as *low-voltage differential signaling* (LVDS). In a typical implementation, the transmitter injects a small current into one wire or the other, depending on the logic level to be sent. The current passes through a resistor at the receiving end, and then returns in the opposite direction along the other wire. The receiver senses the polarity of the voltage across the resistor to determine the logic level.

Another function performed by the physical layer is that it manages the translation between 80-bit flits and 20-bit phits using a technique known as **multilane distribution**. The flits can be considered as a bit stream that is distributed across the data lanes in a round-robin fashion (first bit to first lane, second bit to second lane, etc.), as illustrated in Figure 3.20. This approach enables QPI to achieve very high data rates by implementing the physical link between two ports as multiple parallel channels.

QPI Link Layer

The QPI link layer performs two key functions: flow control and error control. These functions are performed as part of the QPI link layer protocol, and operate on the level of the flit (flow control unit). Each flit consists of a 72-bit message payload and

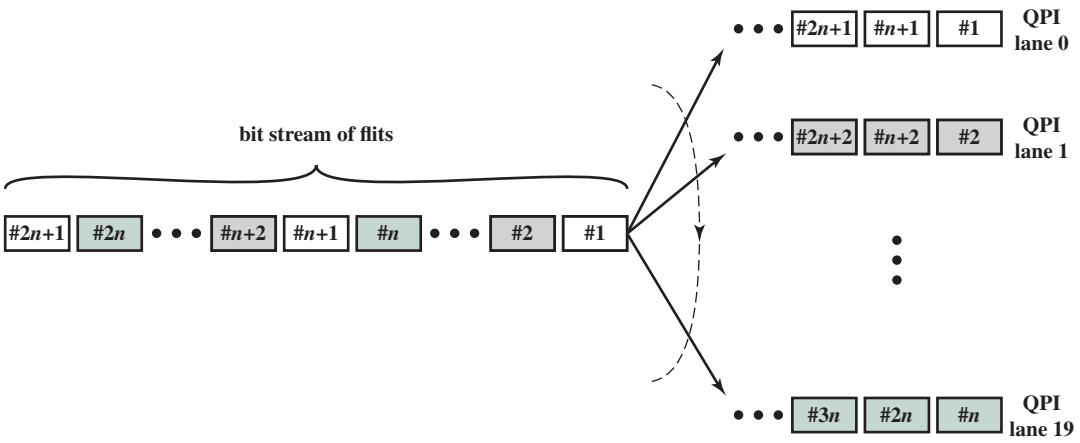


Figure 3.20 QPI Multilane Distribution

an 8-bit error control code called a cyclic redundancy check (CRC). We discuss error control codes in Chapter 5.

A flit payload may consist of data or message information. The data flits transfer the actual bits of data between cores or between a core and an IOH. The message flits are used for such functions as flow control, error control, and cache coherence. We discuss cache coherence in Chapters 5 and 17.

The **flow control function** is needed to ensure that a sending QPI entity does not overwhelm a receiving QPI entity by sending data faster than the receiver can process the data and clear buffers for more incoming data. To control the flow of data, QPI makes use of a credit scheme. During initialization, a sender is given a set number of credits to send flits to a receiver. Whenever a flit is sent to the receiver, the sender decrements its credit counters by one credit. Whenever a buffer is freed at the receiver, a credit is returned to the sender for that buffer. Thus, the receiver controls that pace at which data is transmitted over a QPI link.

Occasionally, a bit transmitted at the physical layer is changed during transmission, due to noise or some other phenomenon. The **error control function** at the link layer detects and recovers from such bit errors, and so isolates higher layers from experiencing bit errors. The procedure works as follows for a flow of data from system A to system B:

1. As mentioned, each 80-bit flit includes an 8-bit CRC field. The CRC is a function of the value of the remaining 72 bits. On transmission, A calculates a CRC value for each flit and inserts that value into the flit.
2. When a flit is received, B calculates a CRC value for the 72-bit payload and compares this value with the value of the incoming CRC value in the flit. If the two CRC values do not match, an error has been detected.
3. When B detects an error, it sends a request to A to retransmit the flit that is in error. However, because A may have had sufficient credit to send a stream of flits, so that additional flits have been transmitted after the flit in error and

before A receives the request to retransmit. Therefore, the request is for A to back up and retransmit the damaged flit plus all subsequent flits.

QPI Routing Layer

The routing layer is used to determine the course that a packet will traverse across the available system interconnects. Routing tables are defined by firmware and describe the possible paths that a packet can follow. In small configurations, such as a two-socket platform, the routing options are limited and the routing tables quite simple. For larger systems, the routing table options are more complex, giving the flexibility of routing and rerouting traffic depending on how (1) devices are populated in the platform, (2) system resources are partitioned, and (3) reliability events result in mapping around a failing resource.

QPI Protocol Layer

In this layer, the packet is defined as the unit of transfer. The packet contents definition is standardized with some flexibility allowed to meet differing market segment requirements. One key function performed at this level is a cache coherency protocol, which deals with making sure that main memory values held in multiple caches are consistent. A typical data packet payload is a block of data being sent to or from a cache.

3.6 PCI EXPRESS

The **peripheral component interconnect (PCI)** is a popular high-bandwidth, processor-independent bus that can function as a mezzanine or peripheral bus. Compared with other common bus specifications, PCI delivers better system performance for high-speed I/O subsystems (e.g., graphic display adapters, network interface controllers, and disk controllers).

Intel began work on PCI in 1990 for its Pentium-based systems. Intel soon released all the patents to the public domain and promoted the creation of an industry association, the PCI Special Interest Group (SIG), to develop further and maintain the compatibility of the PCI specifications. The result is that PCI has been widely adopted and is finding increasing use in personal computer, workstation, and server systems. Because the specification is in the public domain and is supported by a broad cross-section of the microprocessor and peripheral industry, PCI products built by different vendors are compatible.

As with the system bus discussed in the preceding sections, the bus-based PCI scheme has not been able to keep pace with the data rate demands of attached devices. Accordingly, a new version, known as **PCI Express (PCIe)** has been developed. PCIe, as with QPI, is a point-to-point interconnect scheme intended to replace bus-based schemes such as PCI.

A key requirement for PCIe is high capacity to support the needs of higher data rate I/O devices, such as Gigabit Ethernet. Another requirement deals with the need to support time-dependent data streams. Applications such as video-on-demand and audio redistribution are putting real-time constraints on servers too. Many communications applications and embedded PC control systems also process data in real-time. Today's platforms must also deal with multiple concurrent

transfers at ever-increasing data rates. It is no longer acceptable to treat all data as equal—it is more important, for example, to process streaming data first since late real-time data is as useless as no data. Data needs to be tagged so that an I/O system can prioritize its flow throughout the platform.

PCI Physical and Logical Architecture

Figure 3.21 shows a typical configuration that supports the use of PCIe. A **root complex** device, also referred to as a *chipset* or a *host bridge*, connects the processor and memory subsystem to the PCI Express switch fabric comprising one or more PCIe and PCIe switch devices. The root complex acts as a buffering device, to deal with difference in data rates between I/O controllers and memory and processor components. The root complex also translates between PCIe transaction formats and the processor and memory signal and control requirements. The chipset will typically support multiple PCIe ports, some of which attach directly to a PCIe device, and one or more that attach to a switch that manages multiple PCIe streams. PCIe links from the chipset may attach to the following kinds of devices that implement PCIe:

- **Switch:** The switch manages multiple PCIe streams.
- **PCIe endpoint:** An I/O device or controller that implements PCIe, such as a Gigabit ethernet switch, a graphics or video controller, disk interface, or a communications controller.

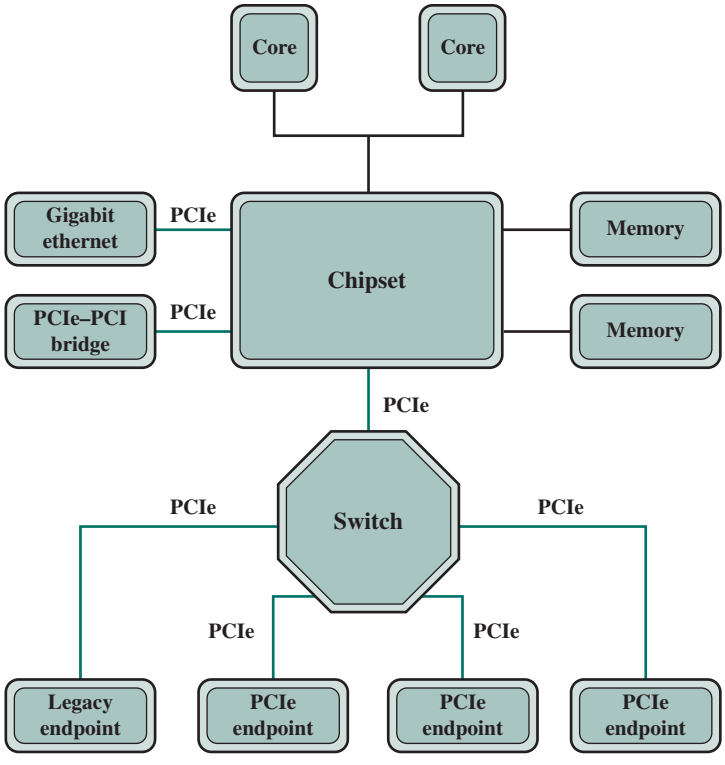


Figure 3.21 Typical Configuration Using PCIe

- **Legacy endpoint:** Legacy endpoint category is intended for existing designs that have been migrated to PCI Express, and it allows legacy behaviors such as use of I/O space and locked transactions. PCI Express endpoints are not permitted to require the use of I/O space at runtime and must not use locked transactions. By distinguishing these categories, it is possible for a system designer to restrict or eliminate legacy behaviors that have negative impacts on system performance and robustness.
- **PCIe/PCI bridge:** Allows older PCI devices to be connected to PCIe-based systems.

As with QPI, PCIe interactions are defined using a protocol architecture. The PCIe protocol architecture encompasses the following layers (Figure 3.22):

- **Physical:** Consists of the actual wires carrying the signals, as well as circuitry and logic to support ancillary features required in the transmission and receipt of the 1s and 0s.
- **Data link:** Is responsible for reliable transmission and flow control. Data packets generated and consumed by the DLL are called Data Link Layer Packets (DLLPs).
- **Transaction:** Generates and consumes data packets used to implement load/store data transfer mechanisms and also manages the flow control of those packets between the two components on a link. Data packets generated and consumed by the TL are called Transaction Layer Packets (TLPs).

Above the TL are software layers that generate read and write requests that are transported by the transaction layer to the I/O devices using a packet-based transaction protocol.

PCIe Physical Layer

Similar to QPI, PCIe is a point-to-point architecture. Each PCIe port consists of a number of bidirectional lanes (note that in QPI, the lane refers to transfer in one direction only). Transfer in each direction in a lane is by means of differential signaling over a pair of wires. A PCI port can provide 1, 4, 6, 16, or 32 lanes. In what follows, we refer to the PCIe 3.0 specification, introduced in late 2010.

As with QPI, PCIe uses a multilane distribution technique. Figure 3.23 shows an example for a PCIe port consisting of four lanes. Data are distributed to the four

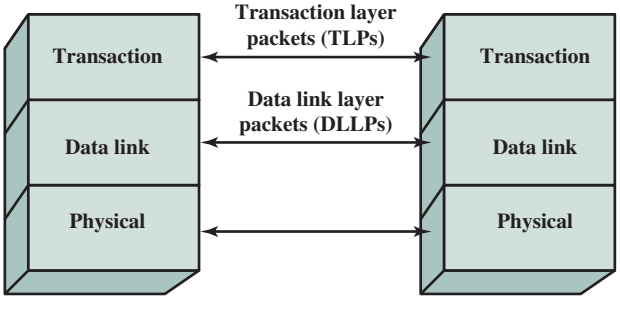


Figure 3.22 PCIe Protocol Layers

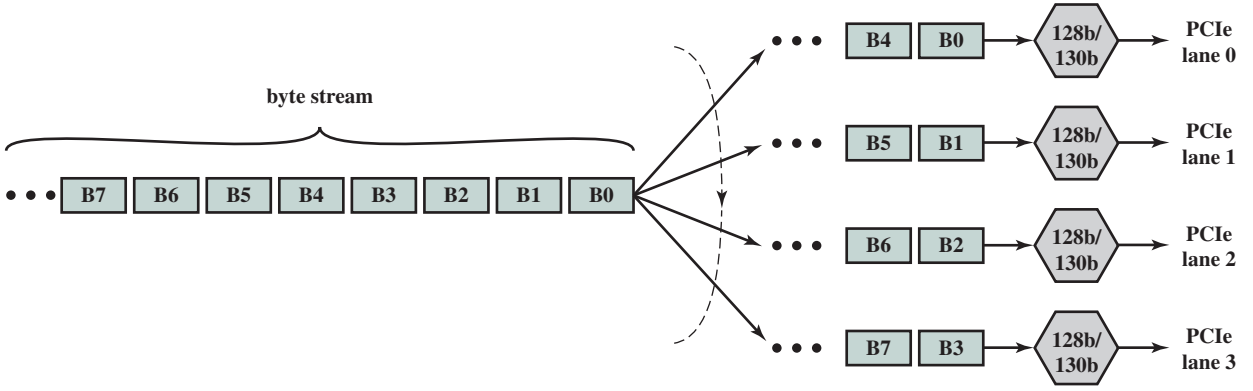


Figure 3.23 PCIe Multilane Distribution

lanes 1 byte at a time using a simple round-robin scheme. At each physical lane, data are buffered and processed 16 bytes (128 bits) at a time. Each block of 128 bits is encoded into a unique 130-bit codeword for transmission; this is referred to as 128b/130b encoding. Thus, the effective data rate of an individual lane is reduced by a factor of 128/130.

To understand the rationale for the 128b/130b encoding, note that unlike QPI, PCIe does not use its clock line to synchronize the bit stream. That is, the clock line is not used to determine the start and end point of each incoming bit; it is used for other signaling purposes only. However, it is necessary for the receiver to be synchronized with the transmitter, so that the receiver knows when each bit begins and ends. If there is any drift between the clocks used for bit transmission and reception of the transmitter and receiver, errors may occur. To compensate for the possibility of drift, PCIe relies on the receiver synchronizing with the transmitter based on the transmitted signal. As with QPI, PCIe uses differential signaling over a pair of wires. Synchronization can be achieved by the receiver looking for transitions in the data and synchronizing its clock to the transition. However, consider that with a long string of 1s or 0s using differential signaling, the output is a constant voltage over a long period of time. Under these circumstances, any drift between the clocks of transmitter and receiver will result in loss of synchronization between the two.

A common approach, and the one used in PCIe 3.0, to overcoming the problem of a long string of bits of one value is scrambling. Scrambling, which does not increase the number of bits to be transmitted, is a mapping technique that tends to make the data appear more random. The scrambling tends to spread out the number of transitions so that they appear at the receiver more uniformly spaced, which is good for synchronization. Also, other transmission properties, such as spectral properties, are enhanced if the data are more nearly of a random nature rather than constant or repetitive. For more discussion of scrambling, see Appendix E.

Another technique that can aid in synchronization is encoding, in which additional bits are inserted into the bit stream to force transitions. For PCIe 3.0, each group of 128 bits of input is mapped into a 130-bit block by adding a 2-bit block sync header. The value of the header is 10 for a data block and 01 for what is called an *ordered set block*, which refers to a link-level information block.

Figure 3.24 illustrates the use of scrambling and encoding. Data to be transmitted are fed into a scrambler. The scrambled output is then fed into a 128b/130b encoder, which buffers 128 bits and then maps the 128-bit block into a 130-bit block. This block then passes through a parallel-to-serial converter and transmitted one bit at a time using differential signaling.

At the receiver, a clock is synchronized to the incoming data to recover the bit stream. This then passes through a serial-to-parallel converter to produce a stream of 130-bit blocks. Each block is passed through a 128b/130b decoder to recover the original scrambled bit pattern, which is then descrambled to produce the original bit stream.

Using these techniques, a data rate of 16 GB/s can be achieved. One final detail to mention; each transmission of a block of data over a PCI link begins and ends with an 8-bit framing sequence intended to give the receiver time to synchronize with the incoming physical layer bit stream.

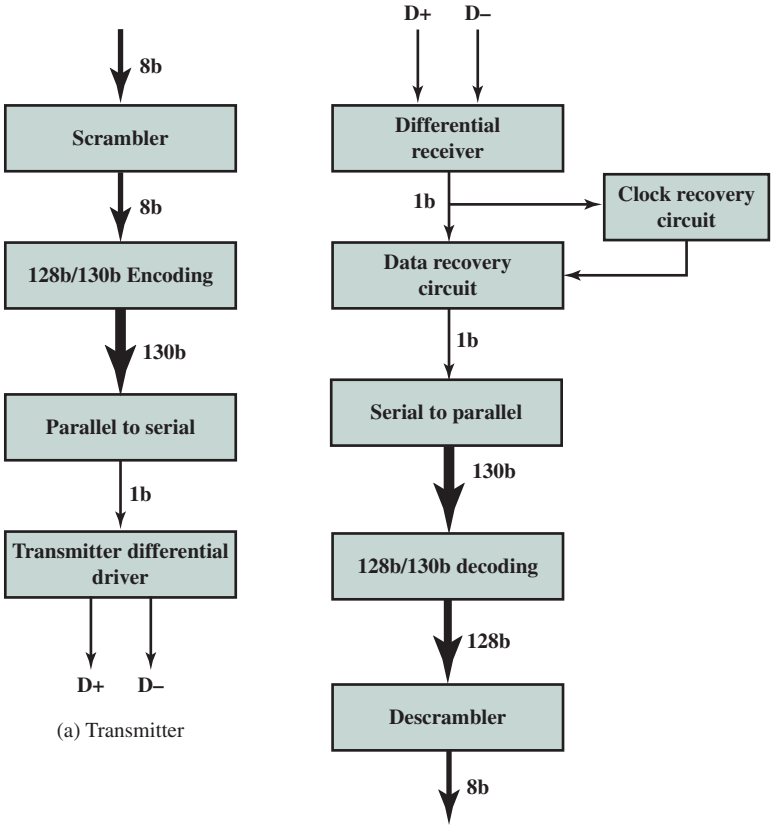


Figure 3.24 PCIe Transmit and Receive Block Diagrams

PCIe Transaction Layer

The transaction layer (TL) receives read and write requests from the software above the TL and creates request packets for transmission to a destination via the link layer. Most transactions use a *split transaction* technique, which works in the following fashion. A request packet is sent out by a source PCIe device, which then waits for a response, called a *completion* packet. The completion following a request is initiated by the completer only when it has the data and/or status ready for delivery. Each packet has a unique identifier that enables completion packets to be directed to the correct originator. With the split transaction technique, the completion is separated in time from the request, in contrast to a typical bus operation in which both sides of a transaction must be available to seize and use the bus. Between the request and the completion, other PCIe traffic may use the link.

TL messages and some write transactions are *posted transactions*, meaning that no response is expected.

The TL packet format supports 32-bit memory addressing and extended 64-bit memory addressing. Packets also have attributes such as "no-snoop,"

"relaxedordering," and "priority," which may be used to optimally route these packets through the I/O subsystem.

ADDRESS SPACES AND TRANSACTION TYPES The TL supports four address spaces:

- **Memory:** The memory space includes system main memory. It also includes PCIe I/O devices. Certain ranges of memory addresses map into I/O devices.
- **I/O:** This address space is used for legacy PCI devices, with reserved memory address ranges used to address legacy I/O devices.
- **Configuration:** This address space enables the TL to read/write configuration registers associated with I/O devices.
- **Message:** This address space is for control signals related to interrupts, error handling, and power management.

Table 3.2 shows the transaction types provided by the TL. For memory, I/O, and configuration address spaces, there are read and write transactions. In the case of memory transactions, there is also a read lock request function. Locked operations occur as a result of device drivers requesting atomic access to registers on a PCIe device. A device driver, for example, can atomically read, modify, and then write to a device register. To accomplish this, the device driver causes the processor to execute an instruction or set of instructions. The root complex converts these processor instructions into a sequence of PCIe transactions, which perform individual read and write requests for the device driver. If these transactions must be executed atomically, the root complex locks the PCIe link while executing the transactions. This locking prevents transactions that are not part of the sequence from occurring. This sequence of transactions is called a locked operation. The particular set

Table 3.2 PCIe TLP Transaction Types

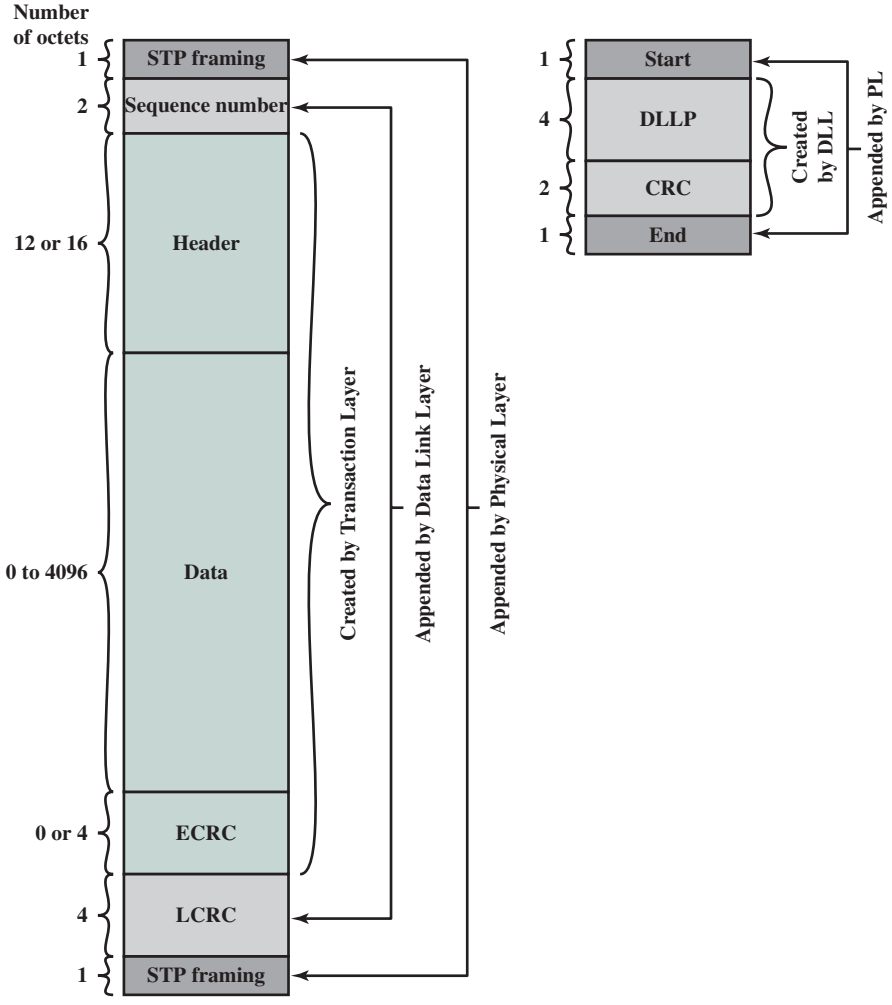
Address Space	TLP Type	Purpose
Memory	Memory Read Request	Transfer data to or from a location in the system memory map.
	Memory Read Lock Request	
	Memory Write Request	
I/O	I/O Read Request	Transfer data to or from a location in the system memory map for legacy devices.
	I/O Write Request	
Configuration	Config Type 0 Read Request	Transfer data to or from a location in the configuration space of a PCIe device.
	Config Type 0 Write Request	
	Config Type 1 Read Request	
	Config Type 1 Write Request	
Message	Message Request	Provides in-band messaging and event reporting.
	Message Request with Data	
Memory, I/O, Configuration	Completion	Returned for certain requests.
	Completion with Data	
	Completion Locked	
	Completion Locked with Data	

of processor instructions that can cause a locked operation to occur depends on the system chip set and processor architecture.

To maintain compatibility with PCI, PCIe supports both Type 0 and Type 1 configuration cycles. A Type 1 cycle propagates downstream until it reaches the bridge interface hosting the bus (link) that the target device resides on. The configuration transaction is converted on the destination link from Type 1 to Type 0 by the bridge.

Finally, completion messages are used with split transactions for memory, I/O, and configuration transactions.

TLP PACKET ASSEMBLY PCIe transactions are conveyed using transaction layer packets, which are illustrated in Figure 3.25a. A TLP originates in the transaction layer of the sending device and terminates at the transaction layer of the receiving device.



(a) Transaction Layer Packet (b) Data Link Layer Packet

Figure 3.25 PCIe Protocol Data Unit Format

Upper layer software sends to the TL the information needed for the TL to create the core of the TLP, which consists of the following fields:

- **Header:** The header describes the type of packet and includes information needed by the receiver to process the packet, including any needed routing information. The internal header format is discussed subsequently.
- **Data:** A data field of up to 4096 bytes may be included in the TLP. Some TLPs do not contain a data field.
- **ECRC:** An optional end-to-end CRC field enables the destination TL layer to check for errors in the header and data portions of the TLP.

PCIe Data Link Layer

The purpose of the PCIe data link layer is to ensure reliable delivery of packets across the PCIe link. The DLL participates in the formation of TLPs and also transmits DLLPs.

DATA LINK LAYER PACKETS Data link layer packets originate at the data link layer of a transmitting device and terminate at the DLL of the device on the other end of the link. Figure 3.25b shows the format of a DLLP. There are three important groups of DLLPs used in managing a link: flow control packets, power management packets, and TLP ACK and NAK packets. Power management packets are used in managing power platform budgeting. Flow control packets regulate the rate at which TLPs and DLLPs can be transmitted across a link. The ACK and NAK packets are used in TLP processing, discussed in the following paragraphs.

TRANSACTION LAYER PACKET PROCESSING The DLL adds two fields to the core of the TLP created by the TL (Figure 3.25a): a 16-bit sequence number and a 32-bit link-layer CRC (LCRC). Whereas the core fields created at the TL are only used at the destination TL, the two fields added by the DLL are processed at each intermediate node on the way from source to destination.

When a TLP arrives at a device, the DLL strips off the sequence number and LCRC fields and checks the LCRC. There are two possibilities:

1. If no errors are detected, the core portion of the TLP is handed up to the local transaction layer. If this receiving device is the intended destination, then the TL processes the TLP. Otherwise, the TL determines a route for the TLP and passes it back down to the DLL for transmission over the next link on the way to the destination.
2. If an error is detected, the DLL schedules an NAK DLL packet to return back to the remote transmitter. The TLP is eliminated.

When the DLL transmits a TLP, it retains a copy of the TLP. If it receives an NAK for the TLP with this sequence number, it retransmits the TLP. When it receives an ACK, it discards the buffered TLP.

3.7 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

address bus	execute cycle	multilane distribution
address lines	fetch cycle	packets
arbitration	flit	PCI Express (PCIe)
balanced transmission	flow control function	peripheral component interconnect (PCI)
bus	instruction cycle	phit
control lines	interrupt	QuickPath Interconnect (QPI)
data bus	interrupt handler	root complex
data lines	interrupt service routine (ISR)	system bus
differential signaling	lane	
disabled interrupt	memory address register (MAR)	
distributed arbitration	memory buffer register (MBR)	
error control function		

Review Questions

3.1 What general categories of functions are specified by computer instructions?
3.2 List and briefly define the possible states that define an instruction execution.
3.3 List and briefly define two approaches to dealing with multiple interrupts.
3.4 What types of transfers must a computer's interconnection structure (e.g., bus) support?
3.5 List and briefly define the QPI protocol layers.
3.6 List and briefly define the PCIe protocol layers.

Problems

3.1 The hypothetical machine of Figure 3.4 also has two I/O instructions:

0011 = Load AC from I/O

0111 = Store AC to I/O

In these cases, the 12-bit address identifies a particular I/O device. Show the program execution (using the format of Figure 3.5) for the following program:

1. Load AC from device 5.
2. Add contents of memory location 940.
3. Store AC to device 6.

Assume that the next value retrieved from device 5 is 3 and that location 940 contains a value of 2.

3.2 The program execution of Figure 3.5 is described in the text using six steps. Expand this description to show the use of the MAR and MBR.

3.3 Consider a hypothetical 32-bit microprocessor having 32-bit instructions composed of two fields: the first byte contains the opcode and the remainder the immediate operand or an operand address.
- **a.** What is the maximum directly addressable memory capacity (in bytes)?
- **b.** Discuss the impact on the system speed if the microprocessor bus has:
  - **1.** 32-bit local address bus and a 16-bit local data bus, or
  - **2.** 16-bit local address bus and a 16-bit local data bus.
- **c.** How many bits are needed for the program counter and the instruction register?

3.4 Consider a hypothetical microprocessor generating a 16-bit address (for example, assume that the program counter and the address registers are 16 bits wide) and having a 16-bit data bus.
- **a.** What is the maximum memory address space that the processor can access directly if it is connected to a "16-bit memory"?
- **b.** What is the maximum memory address space that the processor can access directly if it is connected to an "8-bit memory"?
- **c.** What architectural features will allow this microprocessor to access a separate "I/O space"?
- **d.** If an input and an output instruction can specify an 8-bit I/O port number, how many 8-bit I/O ports can the microprocessor support? How many 16-bit I/O ports? Explain.

3.5 Consider a 32-bit microprocessor, with a 16-bit external data bus, driven by an 8-MHz input clock. Assume that this microprocessor has a bus cycle whose minimum duration equals four input clock cycles. What is the maximum data transfer rate across the bus that this microprocessor can sustain, in bytes/sec? To increase its performance, would it be better to make its external data bus 32 bits or to double the external clock frequency supplied to the microprocessor? State any other assumptions you make, and explain. *Hint:* Determine the number of bytes that can be transferred per bus cycle.

3.6 Consider a computer system that contains an I/O module controlling a simple keyboard/printer teletype. The following registers are contained in the processor and connected directly to the system bus:

INPR: Input Register, 8 bits
OUTR: Output Register, 8 bits
FGI: Input Flag, 1 bit
FGO: Output Flag, 1 bit
IEN: Interrupt Enable, 1 bit

Keystroke input from the teletype and printer output to the teletype are controlled by the I/O module. The teletype is able to encode an alphanumeric symbol to an 8-bit word and decode an 8-bit word into an alphanumeric symbol.
- **a.** Describe how the processor, using the first four registers listed in this problem, can achieve I/O with the teletype.
- **b.** Describe how the function can be performed more efficiently by also employing IEN.

3.7 Consider two microprocessors having 8- and 16-bit-wide external data buses, respectively. The two processors are identical otherwise and their bus cycles take just as long.
- **a.** Suppose all instructions and operands are two bytes long. By what factor do the maximum data transfer rates differ?
- **b.** Repeat assuming that half of the operands and instructions are one byte long.

3.8 Figure 3.26 indicates a distributed arbitration scheme that can be used with an obsolete bus scheme known as Multibus I. Agents are daisy-chained physically in priority order. The left-most agent in the diagram receives a constant *bus priority in* (BPRN) signal indicating that no higher-priority agent desires the bus. If the agent does not require the bus, it asserts its *bus priority out* (BPRO) line. At the beginning of a clock

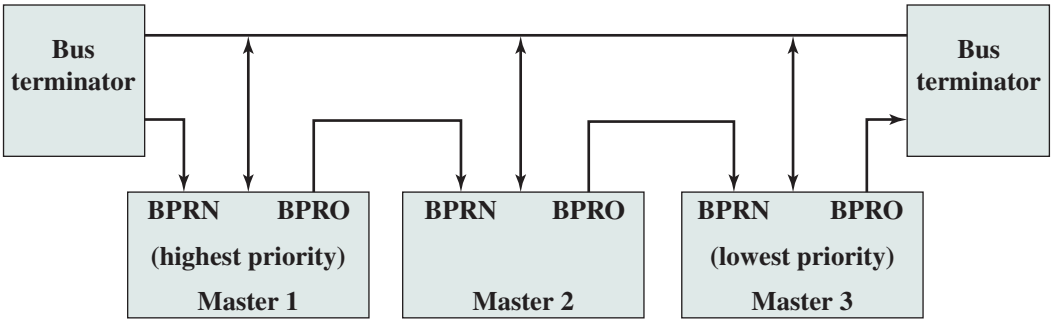


Figure 3.26 Multibus I Distributed Arbitration

cycle, any agent can request control of the bus by lowering its BPRO line. This lowers the BPRN line of the next agent in the chain, which is in turn required to lower its BPRO line. Thus, the signal is propagated the length of the chain. At the end of this chain reaction, there should be only one agent whose BPRN is asserted and whose BPRO is not. This agent has priority. If, at the beginning of a bus cycle, the bus is not busy (BUSY inactive), the agent that has priority may seize control of the bus by asserting the BUSY line.

It takes a certain amount of time for the BPR signal to propagate from the highest-priority agent to the lowest. Must this time be less than the clock cycle? Explain.

3.9 The VAX SBI bus uses a distributed, synchronous arbitration scheme. Each SBI device (i.e., processor, memory, I/O module) has a unique priority and is assigned a unique transfer request (TR) line. The SBI has 16 such lines (TR0, TR1, . . ., TR15), with TR0 having the highest priority. When a device wants to use the bus, it places a reservation for a future time slot by asserting its TR line during the current time slot. At the end of the current time slot, each device with a pending reservation examines the TR lines; the highest-priority device with a reservation uses the next time slot.

A maximum of 17 devices can be attached to the bus. The device with priority 16 has no TR line. Why not?

3.10 On the VAX SBI, the lowest-priority device usually has the lowest average wait time. For this reason, the processor is usually given the lowest priority on the SBI. Why does the priority 16 device usually have the lowest average wait time? Under what circumstances would this not be true?

3.11 For a synchronous read operation (Figure 3.18), the memory module must place the data on the bus sufficiently ahead of the falling edge of the Read signal to allow for signal settling. Assume a microprocessor bus is clocked at 10 MHz and that the Read signal begins to fall in the middle of the second half of T_3.

a. Determine the length of the memory read instruction cycle.

b. When, at the latest, should memory data be placed on the bus? Allow 20 ns for the settling of data lines.

3.12 Consider a microprocessor that has a memory read timing as shown in Figure 3.18. After some analysis, a designer determines that the memory falls short of providing read data on time by about 180 ns.

a. How many wait states (clock cycles) need to be inserted for proper system operation if the bus clocking rate is 8 MHz?

b. To enforce the wait states, a Ready status line is employed. Once the processor has issued a Read command, it must wait until the Ready line is asserted before attempting to read data. At what time interval must we keep the Ready line low in order to force the processor to insert the required number of wait states?

3.13 A microprocessor has a memory write timing as shown in Figure 3.18. Its manufacturer specifies that the width of the Write signal can be determined by $T-50$, where T is the clock period in ns.
- **a.** What width should we expect for the Write signal if bus clocking rate is 5 MHz?
- **b.** The data sheet for the microprocessor specifies that the data remain valid for 20 ns after the falling edge of the Write signal. What is the total duration of valid data presentation to memory?
- **c.** How many wait states should we insert if memory requires valid data presentation for at least 190 ns?

3.14 A microprocessor has an increment memory direct instruction, which adds 1 to the value in a memory location. The instruction has five stages: fetch opcode (four bus clock cycles), fetch operand address (three cycles), fetch operand (three cycles), add 1 to operand (three cycles), and store operand (three cycles).
- **a.** By what amount (in percent) will the duration of the instruction increase if we have to insert two bus wait states in each memory read and memory write operation?
- **b.** Repeat assuming that the increment operation takes 13 cycles instead of 3 cycles.

3.15 The Intel 8088 microprocessor has a read bus timing similar to that of Figure 3.18, but requires four processor clock cycles. The valid data is on the bus for an amount of time that extends into the fourth processor clock cycle. Assume a processor clock rate of 8 MHz.
- **a.** What is the maximum data transfer rate?
- **b.** Repeat, but assume the need to insert one wait state per byte transferred.

3.16 The Intel 8086 is a 16-bit processor similar in many ways to the 8-bit 8088. The 8086 uses a 16-bit bus that can transfer 2 bytes at a time, provided that the lower-order byte has an even address. However, the 8086 allows both even- and odd-aligned word operands. If an odd-aligned word is referenced, two memory cycles, each consisting of four bus cycles, are required to transfer the word. Consider an instruction on the 8086 that involves two 16-bit operands. How long does it take to fetch the operands? Give the range of possible answers. Assume a clocking rate of 4 MHz and no wait states.

3.17 Consider a 32-bit microprocessor whose bus cycle is the same duration as that of a 16-bit microprocessor. Assume that, on average, 20% of the operands and instructions are 32 bits long, 40% are 16 bits long, and 40% are only 8 bits long. Calculate the improvement achieved when fetching instructions and operands with the 32-bit microprocessor.

3.18 The microprocessor of Problem 3.14 initiates the fetch operand stage of the increment memory direct instruction at the same time that a keyboard actives an interrupt request line. After how long does the processor enter the interrupt processing cycle? Assume a bus clocking rate of 10 MHz.

CHAPTER 4

CACHE MEMORY

4.1 Computer Memory System Overview
- Characteristics of Memory Systems
- The Memory Hierarchy

4.2 Cache Memory Principles

4.3 Elements of Cache Design
- Cache Addresses
- Cache Size
- Mapping Function
- Replacement Algorithms
- Write Policy
- Line Size
- Number of Caches

4.4 Pentium 4 Cache Organization

4.5 Key Terms, Review Questions, and Problems

Appendix 4A Performance Characteristics of Two-Level Memories
- Locality
- Operation of Two-Level Memory
- Performance

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Present an overview of the main characteristics of computer memory systems and the use of a memory hierarchy.
- Describe the basic concepts and intent of cache memory.
- Discuss the key elements of cache design.
- Distinguish among direct mapping, associative mapping, and set-associative mapping.
- Explain the reasons for using multiple levels of cache.
- Understand the performance implications of multiple levels of memory.

Although seemingly simple in concept, computer memory exhibits perhaps the widest range of type, technology, organization, performance, and cost of any feature of a computer system. No single technology is optimal in satisfying the memory requirements for a computer system. As a consequence, the typical computer system is equipped with a hierarchy of memory subsystems, some internal to the system (directly accessible by the processor) and some external (accessible by the processor via an I/O module).

This chapter and the next focus on internal memory elements, while Chapter 6 is devoted to external memory. To begin, the first section examines key characteristics of computer memories. The remainder of the chapter examines an essential element of all modern computer systems: cache memory.

4.1 COMPUTER MEMORY SYSTEM OVERVIEW

Characteristics of Memory Systems

The complex subject of computer memory is made more manageable if we classify memory systems according to their key characteristics. The most important of these are listed in Table 4.1.

The term **location** in Table 4.1 refers to whether memory is internal or external to the computer. Internal memory is often equated with main memory, but there are other forms of internal memory. The processor requires its own local memory, in the form of registers (e.g., see Figure 2.3). Further, as we will see, the control unit portion of the processor may also require its own internal memory. We will defer discussion of these latter two types of internal memory to later chapters. Cache is another form of internal memory. External memory consists of peripheral storage devices, such as disk and tape, that are accessible to the processor via I/O controllers.

An obvious characteristic of memory is its **capacity**. For internal memory, this is typically expressed in terms of bytes (1 byte = 8 bits) or words. Common word lengths are 8, 16, and 32 bits. External memory capacity is typically expressed in terms of bytes.

Table 4.1 Key Characteristics of Computer Memory Systems

Location	**Performance**
Internal (e.g., processor registers, cache, main memory)	Access time
External (e.g., optical disks, magnetic disks, tapes)	Cycle time
Capacity	Transfer rate
Number of words	**Physical Type**
Number of bytes	Semiconductor
Unit of Transfer	Magnetic
Word	Optical
Block	Magneto-optical
Access Method	**Physical Characteristics**
Sequential	Volatile/nonvolatile
Direct	Erasable/nonerasable
Random	**Organization**
Associative	Memory modules

A related concept is the **unit of transfer**. For internal memory, the unit of transfer is equal to the number of electrical lines into and out of the memory module. This may be equal to the word length, but is often larger, such as 64, 128, or 256 bits. To clarify this point, consider three related concepts for internal memory:

- **Word:** The "natural" unit of organization of memory. The size of a word is typically equal to the number of bits used to represent an integer and to the instruction length. Unfortunately, there are many exceptions. For example, the CRAY C90 (an older model CRAY supercomputer) has a 64-bit word length but uses a 46-bit integer representation. The Intel x86 architecture has a wide variety of instruction lengths, expressed as multiples of bytes, and a word size of 32 bits.
- **Addressable units:** In some systems, the addressable unit is the word. However, many systems allow addressing at the byte level. In any case, the relationship between the length in bits A of an address and the number N of addressable units is $2^A = N$.
- **Unit of transfer:** For main memory, this is the number of bits read out of or written into memory at a time. The unit of transfer need not equal a word or an addressable unit. For external memory, data are often transferred in much larger units than a word, and these are referred to as blocks.

Another distinction among memory types is the **method of accessing** units of data. These include the following:

- **Sequential access:** Memory is organized into units of data, called records. Access must be made in a specific linear sequence. Stored addressing information is used to separate records and assist in the retrieval process. A shared read–write mechanism is used, and this must be moved from its current location to the desired location, passing and rejecting each intermediate record. Thus, the time to access an arbitrary record is highly variable. Tape units, discussed in Chapter 6, are sequential access.
- **Direct access:** As with sequential access, direct access involves a shared read–write mechanism. However, individual blocks or records have a unique

address based on physical location. Access is accomplished by direct access to reach a general vicinity plus sequential searching, counting, or waiting to reach the final location. Again, access time is variable. Disk units, discussed in Chapter 6, are direct access.

- **Random access:** Each addressable location in memory has a unique, physically wired-in addressing mechanism. The time to access a given location is independent of the sequence of prior accesses and is constant. Thus, any location can be selected at random and directly addressed and accessed. Main memory and some cache systems are random access.
- **Associative:** This is a random access type of memory that enables one to make a comparison of desired bit locations within a word for a specified match, and to do this for all words simultaneously. Thus, a word is retrieved based on a portion of its contents rather than its address. As with ordinary random-access memory, each location has its own addressing mechanism, and retrieval time is constant independent of location or prior access patterns. Cache memories may employ associative access.

From a user's point of view, the two most important characteristics of memory are capacity and **performance**. Three performance parameters are used:

- **Access time (latency):** For random-access memory, this is the time it takes to perform a read or write operation, that is, the time from the instant that an address is presented to the memory to the instant that data have been stored or made available for use. For non-random-access memory, access time is the time it takes to position the read–write mechanism at the desired location.
- **Memory cycle time:** This concept is primarily applied to random-access memory and consists of the access time plus any additional time required before a second access can commence. This additional time may be required for transients to die out on signal lines or to regenerate data if they are read destructively. Note that memory cycle time is concerned with the system bus, not the processor.
- **Transfer rate:** This is the rate at which data can be transferred into or out of a memory unit. For random-access memory, it is equal to 1/(cycle time). For non-random-access memory, the following relationship holds:

$$T_n = T_A + \frac{n}{R} \tag{4.1}$$

where

T_n = Average time to read or write n bits

T_A = Average access time

n = Number of bits

R = Transfer rate, in bits per second (bps)

A variety of **physical types** of memory have been employed. The most common today are semiconductor memory, magnetic surface memory, used for disk and tape, and optical and magneto-optical.

Several **physical characteristics** of data storage are important. In a volatile memory, information decays naturally or is lost when electrical power is switched off. In a nonvolatile memory, information once recorded remains without deterioration until deliberately changed; no electrical power is needed to retain information. Magnetic-surface memories are nonvolatile. Semiconductor memory (memory on integrated circuits) may be either volatile or nonvolatile. Nonerasable memory cannot be altered, except by destroying the storage unit. Semiconductor memory of this type is known as *read-only memory* (ROM). Of necessity, a practical nonerasable memory must also be nonvolatile.

For random-access memory, the **organization** is a key design issue. In this context, *organization* refers to the physical arrangement of bits to form words. The obvious arrangement is not always used, as is explained in Chapter 5.

The Memory Hierarchy

The design constraints on a computer's memory can be summed up by three questions: How much? How fast? How expensive?

The question of how much is somewhat open ended. If the capacity is there, applications will likely be developed to use it. The question of how fast is, in a sense, easier to answer. To achieve greatest performance, the memory must be able to keep up with the processor. That is, as the processor is executing instructions, we would not want it to have to pause waiting for instructions or operands. The final question must also be considered. For a practical system, the cost of memory must be reasonable in relationship to other components.

As might be expected, there is a trade-off among the three key characteristics of memory: capacity, access time, and cost. A variety of technologies are used to implement memory systems, and across this spectrum of technologies, the following relationships hold:

- Faster access time, greater cost per bit;
- Greater capacity, smaller cost per bit;
- Greater capacity, slower access time.

The dilemma facing the designer is clear. The designer would like to use memory technologies that provide for large-capacity memory, both because the capacity is needed and because the cost per bit is low. However, to meet performance requirements, the designer needs to use expensive, relatively lower-capacity memories with short access times.

The way out of this dilemma is not to rely on a single memory component or technology, but to employ a **memory hierarchy**. A typical hierarchy is illustrated in Figure 4.1. As one goes down the hierarchy, the following occur:

a. Decreasing cost per bit;
b. Increasing capacity;
c. Increasing access time;
d. Decreasing frequency of access of the memory by the processor.

Thus, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories. The key to the success of this organization

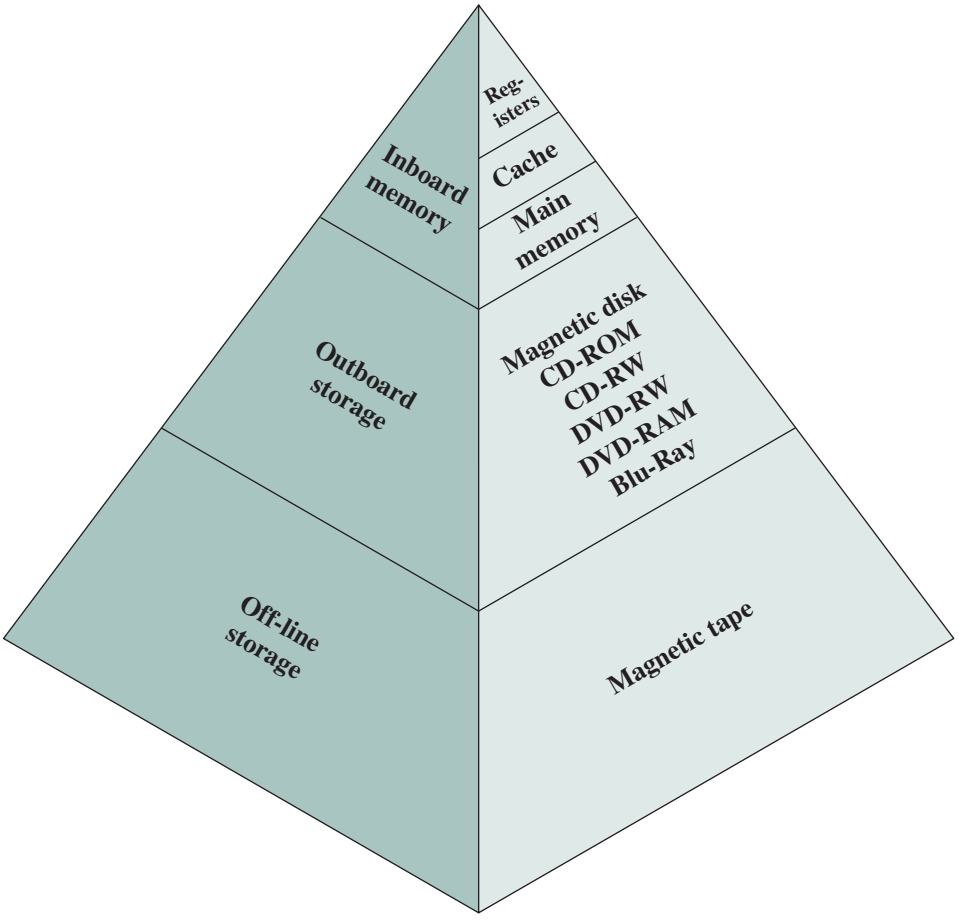


Figure 4.1 The Memory Hierarchy

is item (d): decreasing frequency of access. We examine this concept in greater detail when we discuss the cache, later in this chapter, and virtual memory in Chapter 8. A brief explanation is provided at this point.

The use of two levels of memory to reduce average access time works in principle, but only if conditions (a) through (d) apply. By employing a variety of technologies, a spectrum of memory systems exists that satisfies conditions (a) through (c). Fortunately, condition (d) is also generally valid.

The basis for the validity of condition (d) is a principle known as **locality of reference** [DENN68]. During the course of execution of a program, memory references by the processor, for both instructions and data, tend to cluster. Programs typically contain a number of iterative loops and subroutines. Once a loop or subroutine is entered, there are repeated references to a small set of instructions. Similarly, operations on tables and arrays involve access to a clustered set of data words. Over a long period of time, the clusters in use change, but over a short period of time, the processor is primarily working with fixed clusters of memory references.

EXAMPLE 4.1 Suppose that the processor has access to two levels of memory. Level 1 contains 1000 words and has an access time of 0.01 μs; level 2 contains 100,000 words and has an access time of 0.1 μs. Assume that if a word to be accessed is in level 1, then the processor accesses it directly. If it is in level 2, then the word is first transferred to level 1 and then accessed by the processor. For simplicity, we ignore the time required for the processor to determine whether the word is in level 1 or level 2. Figure 4.2 shows the general shape of the curve that covers this situation. The figure shows the average access time to a two-level memory as a function of the hit ratio H, where H is defined as the fraction of all memory accesses that are found in the faster memory (e.g., the cache), T_1 is the access time to level 1, and T_2 is the access time to level 2.[1] As can be seen, for high percentages of level 1 access, the average total access time is much closer to that of level 1 than that of level 2.

In our example, suppose 95% of the memory accesses are found in level 1. Then the average time to access a word can be expressed as

$$(0.95)(0.01\ \mu s) + (0.05)(0.01\ \mu s + 0.1\ \mu s) = 0.0095 + 0.0055 = 0.015\ \mu s$$

The average access time is much closer to 0.01 μs than to 0.1 μs, as desired.

Accordingly, it is possible to organize data across the hierarchy such that the percentage of accesses to each successively lower level is substantially less than that of the level above. Consider the two-level example already presented. Let level 2

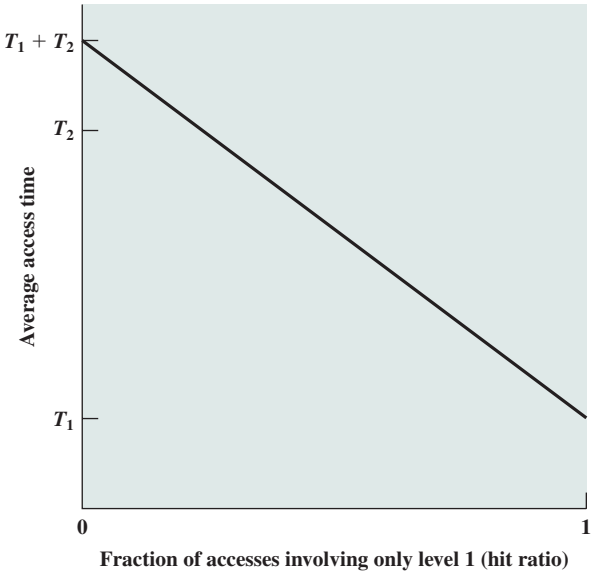


Figure 4.2 Performance of Accesses Involving only Level 1 (hit ratio)

[1] If the accessed word is found in the faster memory, that is defined as a **hit**. A **miss** occurs if the accessed word is not found in the faster memory.

memory contain all program instructions and data. The current clusters can be temporarily placed in level 1. From time to time, one of the clusters in level 1 will have to be swapped back to level 2 to make room for a new cluster coming in to level 1. On average, however, most references will be to instructions and data contained in level 1.

This principle can be applied across more than two levels of memory, as suggested by the hierarchy shown in Figure 4.1. The fastest, smallest, and most expensive type of memory consists of the registers internal to the processor. Typically, a processor will contain a few dozen such registers, although some machines contain hundreds of registers. Main memory is the principal internal memory system of the computer. Each location in main memory has a unique address. Main memory is usually extended with a higher-speed, smaller cache. The cache is not usually visible to the programmer or, indeed, to the processor. It is a device for staging the movement of data between main memory and processor registers to improve performance.

The three forms of memory just described are, typically, volatile and employ semiconductor technology. The use of three levels exploits the fact that semiconductor memory comes in a variety of types, which differ in speed and cost. Data are stored more permanently on external mass storage devices, of which the most common are hard disk and removable media, such as removable magnetic disk, tape, and optical storage. External, nonvolatile memory is also referred to as **secondary memory** or **auxiliary memory**. These are used to store program and data files and are usually visible to the programmer only in terms of files and records, as opposed to individual bytes or words. Disk is also used to provide an extension to main memory known as virtual memory, which is discussed in Chapter 8.

Other forms of memory may be included in the hierarchy. For example, large IBM mainframes include a form of internal memory known as expanded storage. This uses a semiconductor technology that is slower and less expensive than that of main memory. Strictly speaking, this memory does not fit into the hierarchy but is a side branch: Data can be moved between main memory and expanded storage but not between expanded storage and external memory. Other forms of secondary memory include optical and magneto-optical disks. Finally, additional levels can be effectively added to the hierarchy in software. A portion of main memory can be used as a buffer to hold data temporarily that is to be read out to disk. Such a technique, sometimes referred to as a disk cache,[2] improves performance in two ways:

- Disk writes are clustered. Instead of many small transfers of data, we have a few large transfers of data. This improves disk performance and minimizes processor involvement.
- Some data destined for write-out may be referenced by a program before the next dump to disk. In that case, the data are retrieved rapidly from the software cache rather than slowly from the disk.

Appendix 4A examines the performance implications of multilevel memory structures.

[2] Disk cache is generally a purely software technique and is not examined in this book. See [STAL15] for a discussion.

4.2 CACHE MEMORY PRINCIPLES

Cache memory is designed to combine the memory access time of expensive, high-speed memory combined with the large memory size of less expensive, lower-speed memory. The concept is illustrated in Figure 4.3a. There is a relatively large and slow main memory together with a smaller, faster cache memory. The cache contains a copy of portions of main memory. When the processor attempts to read a word of memory, a check is made to determine if the word is in the cache. If so, the word is delivered to the processor. If not, a block of main memory, consisting of some fixed number of words, is read into the cache and then the word is delivered to the processor. Because of the phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single memory reference, it is likely that there will be future references to that same memory location or to other words in the block.

Figure 4.3b depicts the use of multiple levels of cache. The L2 cache is slower and typically larger than the L1 cache, and the L3 cache is slower and typically larger than the L2 cache.

Figure 4.4 depicts the structure of a cache/main-memory system. Main memory consists of up to 2^n addressable words, with each word having a unique n-bit address. For mapping purposes, this memory is considered to consist of a number of fixed-length blocks of K words each. That is, there are $M = 2^n/K$ blocks in main memory. The cache consists of m blocks, called **lines**.[3] Each line contains K words,

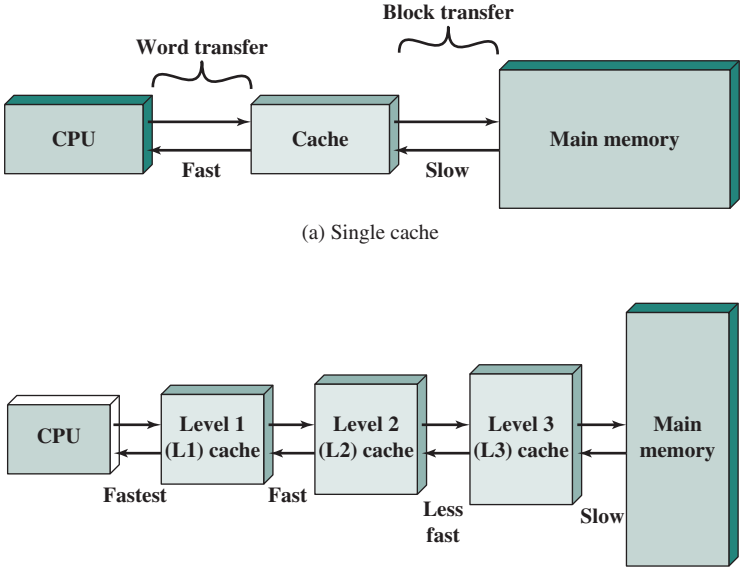


Figure 4.3 Cache and Main Memory

[3]In referring to the basic unit of the cache, the term *line* is used, rather than the term *block*, for two reasons: (1) to avoid confusion with a main memory block, which contains the same number of data words as a cache line; and (2) because a cache line includes not only K words of data, just as a main memory block, but also includes tag and control bits.

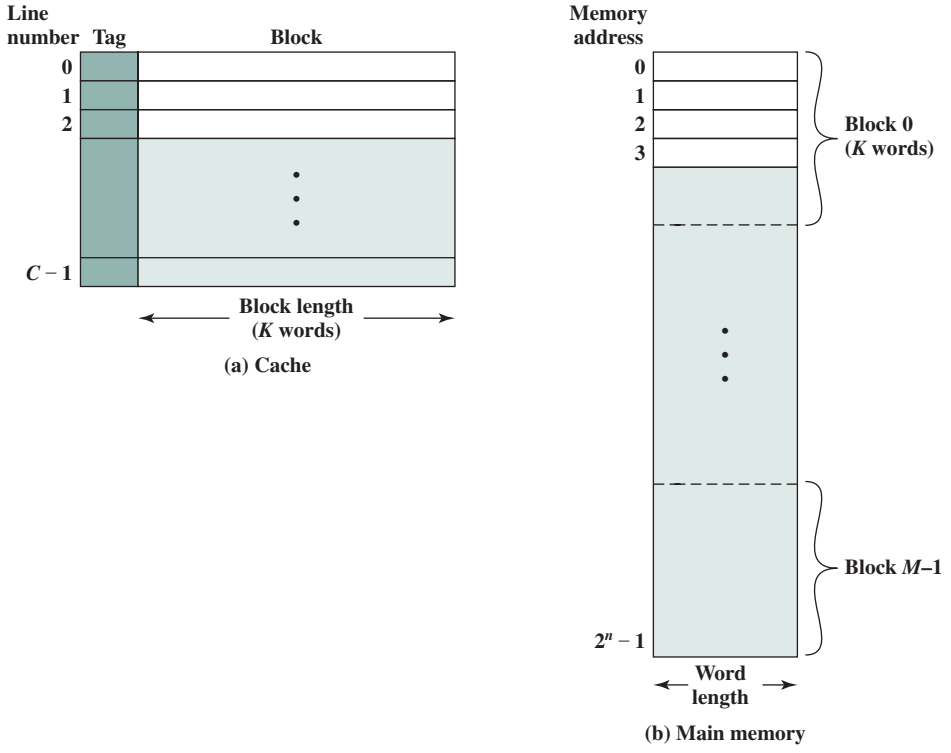


Figure 4.4 Cache/Main Memory Structure

plus a tag of a few bits. Each line also includes control bits (not shown), such as a bit to indicate whether the line has been modified since being loaded into the cache. The length of a line, not including tag and control bits, is the **line size**. The line size may be as small as 32 bits, with each "word" being a single byte; in this case the line size is 4 bytes. The number of lines is considerably less than the number of main memory blocks ($m \ll M$). At any time, some subset of the blocks of memory resides in lines in the cache. If a word in a block of memory is read, that block is transferred to one of the lines of the cache. Because there are more blocks than lines, an individual line cannot be uniquely and permanently dedicated to a particular block. Thus, each line includes a **tag** that identifies which particular block is currently being stored. The tag is usually a portion of the main memory address, as described later in this section.

Figure 4.5 illustrates the read operation. The processor generates the read address (RA) of a word to be read. If the word is contained in the cache, it is delivered to the processor. Otherwise, the block containing that word is loaded into the cache, and the word is delivered to the processor. Figure 4.5 shows these last two operations occurring in parallel and reflects the organization shown in Figure 4.6, which is typical of contemporary cache organizations. In this organization, the cache connects to the processor via data, control, and address lines. The data and address lines also attach to data and address buffers, which attach to a system bus from

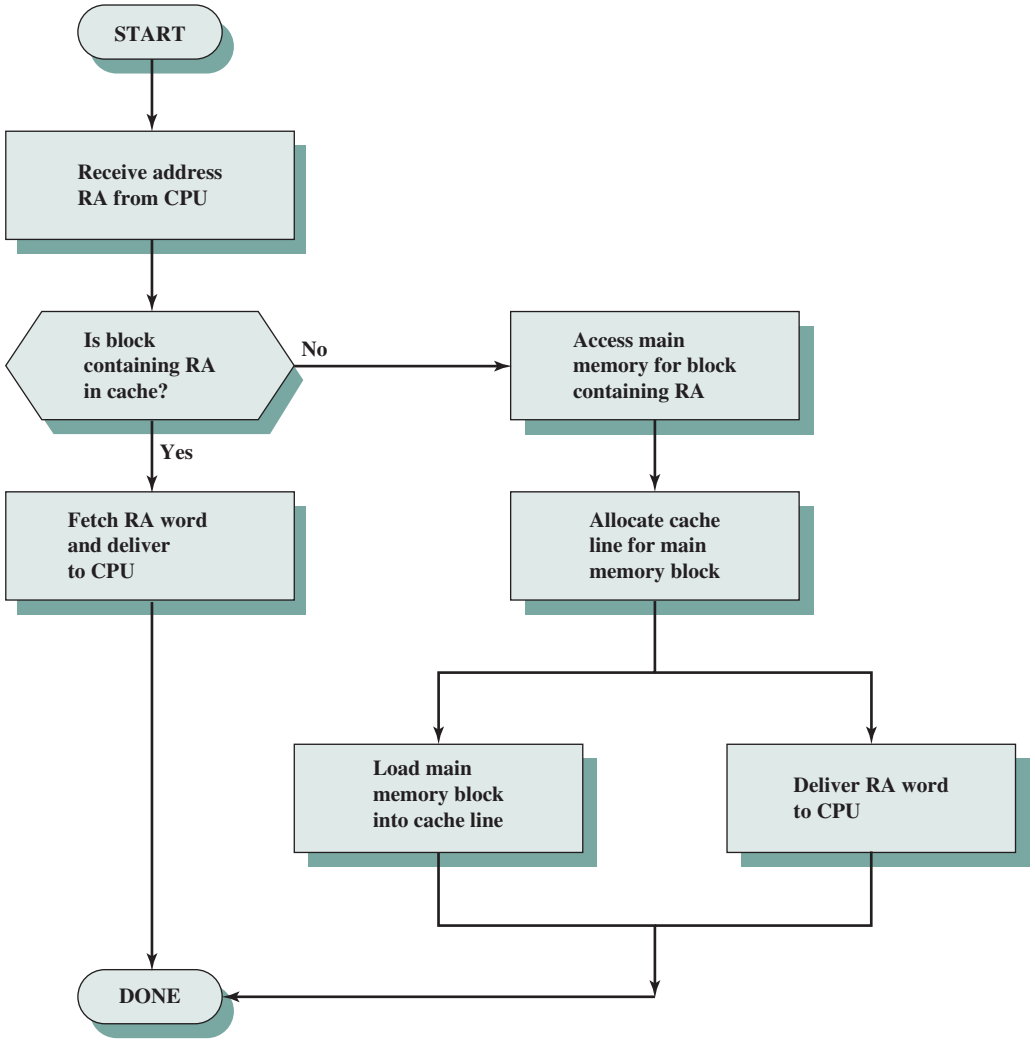

START
Receive address RA from CPU
Is block containing RA in cache?
No
Access main memory for block containing RA
Yes
Fetch RA word and deliver to CPU
Allocate cache line for main memory block
Load main memory block into cache line
Deliver RA word to CPU
DONE

Figure 4.5 Cache Read Operation

which main memory is reached. When a cache hit occurs, the data and address buffers are disabled and communication is only between processor and cache, with no system bus traffic. When a cache miss occurs, the desired address is loaded onto the system bus and the data are returned through the data buffer to both the cache and the processor. In other organizations, the cache is physically interposed between the processor and the main memory for all data, address, and control lines. In this latter case, for a cache miss, the desired word is first read into the cache and then transferred from cache to processor.

A discussion of the performance parameters related to cache use is contained in Appendix 4A.

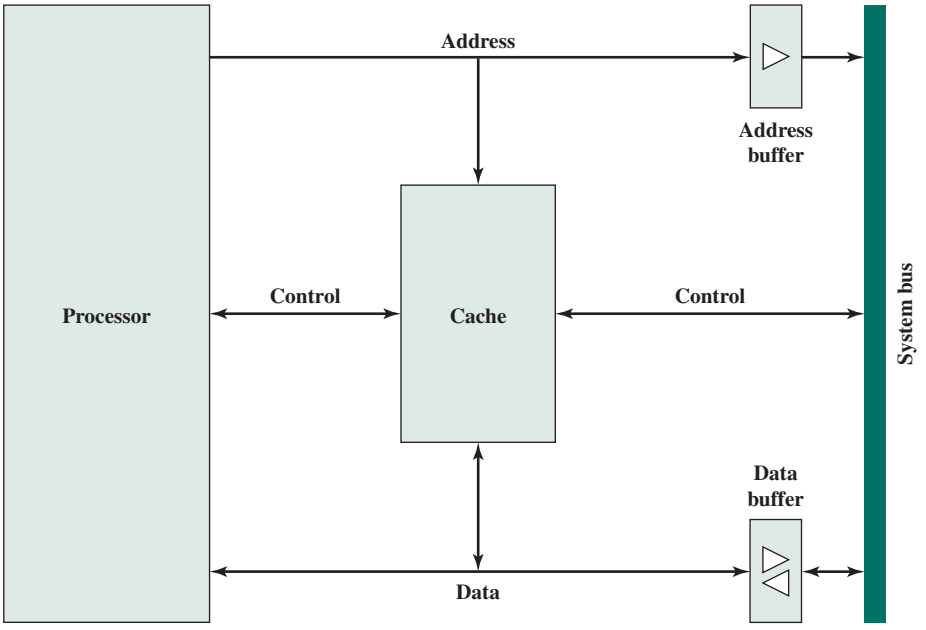


Figure 4.6 Typical Cache Organization

4.3 ELEMENTS OF CACHE DESIGN

This section provides an overview of cache design parameters and reports some typical results. We occasionally refer to the use of caches in **high-performance computing (HPC)**. HPC deals with supercomputers and their software, especially for scientific applications that involve large amounts of data, vector and matrix computation, and the use of parallel algorithms. Cache design for HPC is quite different than for other hardware platforms and applications. Indeed, many researchers have found that HPC applications perform poorly on computer architectures that employ caches [BAIL93]. Other researchers have since shown that a cache hierarchy can be useful in improving performance if the application software is tuned to exploit the cache [WANG99, PRES01].[4]

Although there are a large number of cache implementations, there are a few basic design elements that serve to classify and differentiate cache architectures. Table 4.2 lists key elements.

Cache Addresses

Almost all nonembedded processors, and many embedded processors, support virtual memory, a concept discussed in Chapter 8. In essence, virtual memory is a facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available. When virtual memory is used, the address fields of machine instructions contain virtual addresses. For reads

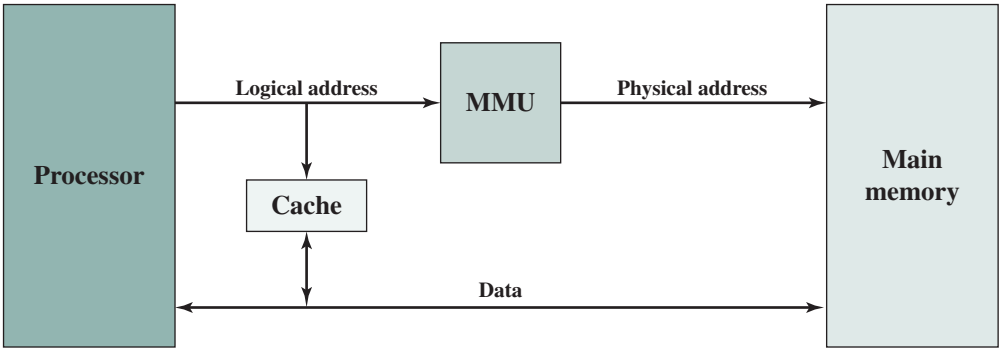
[4]For a general discussion of HPC, see [DOWD98].

Table 4.2 Elements of Cache Design

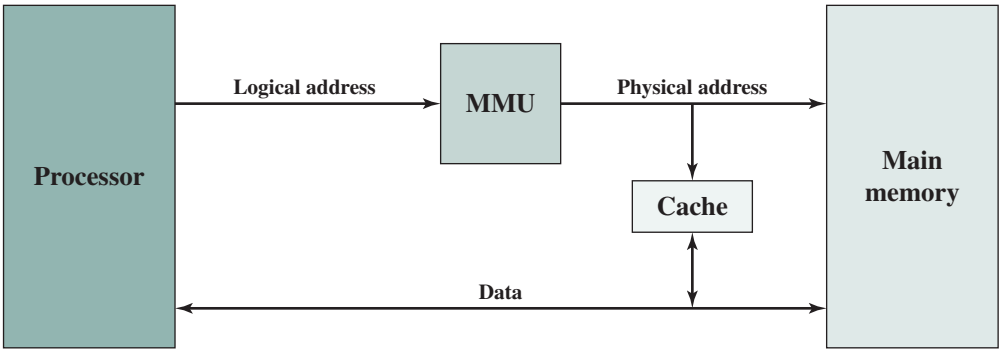
Cache Addresses	**Write Policy**
Logical	Write through
Physical	Write back
Cache Size	**Line Size**
Mapping Function	**Number of Caches**
Direct	Single or two level
Associative	Unified or split
Set associative	
Replacement Algorithm	
Least recently used (LRU)	
First in first out (FIFO)	
Least frequently used (LFU)	
Random	

to and writes from main memory, a hardware memory management unit (MMU) translates each virtual address into a physical address in main memory.

When virtual addresses are used, the system designer may choose to place the cache between the processor and the MMU or between the MMU and main memory (Figure 4.7). A **logical cache**, also known as a **virtual cache**, stores data using



(a) Logical cache



(b) Physical cache

Figure 4.7 Logical and Physical Caches

virtual addresses. The processor accesses the cache directly, without going through the MMU. A **physical cache** stores data using main memory **physical addresses**.

One obvious advantage of the logical cache is that cache access speed is faster than for a physical cache, because the cache can respond before the MMU performs an address translation. The disadvantage has to do with the fact that most virtual memory systems supply each application with the same virtual memory address space. That is, each application sees a virtual memory that starts at address 0. Thus, the same virtual address in two different applications refers to two different physical addresses. The cache memory must therefore be completely flushed with each application context switch, or extra bits must be added to each line of the cache to identify which virtual address space this address refers to.

The subject of logical versus physical cache is a complex one, and beyond the scope of this book. For a more in-depth discussion, see [CEKL97] and [JACO08].

Cache Size

The second item in Table 4.2, cache size, has already been discussed. We would like the size of the cache to be small enough so that the overall average cost per bit is close to that of main memory alone and large enough so that the overall average access time is close to that of the cache alone. There are several other motivations for minimizing cache size. The larger the cache, the larger the number of gates involved in addressing the cache. The result is that large caches tend to be slightly slower than small ones—even when built with the same integrated circuit technology and put in the same place on chip and circuit board. The available chip and board area also limits cache size. Because the performance of the cache is very sensitive to the nature of the workload, it is impossible to arrive at a single "optimum" cache size. Table 4.3 lists the cache sizes of some current and past processors.

Mapping Function

Because there are fewer cache lines than main memory blocks, an algorithm is needed for mapping main memory blocks into cache lines. Further, a means is needed for determining which main memory block currently occupies a cache line. The choice of the mapping function dictates how the cache is organized. Three techniques can be used: direct, associative, and set-associative. We examine each of these in turn. In each case, we look at the general structure and then a specific example.

EXAMPLE 4.2 For all three cases, the example includes the following elements:

- The cache can hold 64 kB.
- Data are transferred between main memory and the cache in blocks of 4 bytes each. This means that the cache is organized as 16K $= 2^{14}$ lines of 4 bytes each.
- The main memory consists of 16 MB, with each byte directly addressable by a 24-bit address ($2^{24} = 16$M). Thus, for mapping purposes, we can consider main memory to consist of 4M blocks of 4 bytes each.

Table 4.3 Cache Sizes of Some Processors

Processor	Type	Year of Introduction	L1 Cache[a]	L2 Cache	L3 Cache
IBM 360/85	Mainframe	1968	16–32 kB	—	—
PDP-11/70	Minicomputer	1975	1 kB	—	—
VAX 11/780	Minicomputer	1978	16 kB	—	—
IBM 3033	Mainframe	1978	64 kB	—	—
IBM 3090	Mainframe	1985	128–256 kB	—	—
Intel 80486	PC	1989	8 kB	—	—
Pentium	PC	1993	8 kB/8 kB	256–512 kB	—
PowerPC 601	PC	1993	32 kB	—	—
PowerPC 620	PC	1996	32 kB/32 kB	—	—
PowerPC G4	PC/server	1999	32 kB/32 kB	256 kB to 1 MB	2 MB
IBM S/390 G6	Mainframe	1999	256 kB	8 MB	—
Pentium 4	PC/server	2000	8 kB/8 kB	256 kB	—
IBM SP	High-end server/ supercomputer	2000	64 kB/32 kB	8 MB	—
CRAY MTA[b]	Supercomputer	2000	8 kB	2 MB	—
Itanium	PC/server	2001	16 kB/16 kB	96 kB	4 MB
Itanium 2	PC/server	2002	32 kB	256 kB	6 MB
IBM POWER5	High-end server	2003	64 kB	1.9 MB	36 MB
CRAY XD-1	Supercomputer	2004	64 kB/64 kB	1 MB	—
IBM POWER6	PC/server	2007	64 kB/64 kB	4 MB	32 MB
IBM z10	Mainframe	2008	64 kB/128 kB	3 MB	24–48 MB
Intel Core i7 EE 990	Workstation/ server	2011	6 × 32 kB/ 32 kB	1.5 MB	12 MB
IBM zEnterprise 196	Mainframe/ server	2011	24 × 64 kB/ 128 kB	24 × 1.5 MB	24 MB L3 192 MB L4

Notes: [a] Two values separated by a slash refer to instruction and data caches. [b] Both caches are instruction only; no data caches.

DIRECT MAPPING The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. The mapping is expressed as

$$i = j \text{ modulo } m$$

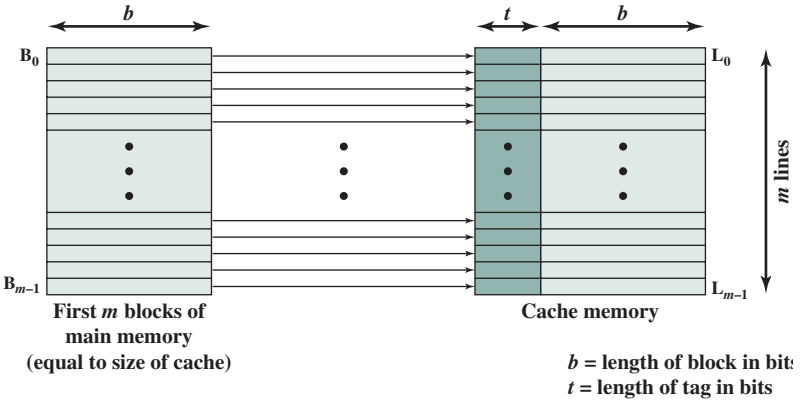
where

i = cache line number

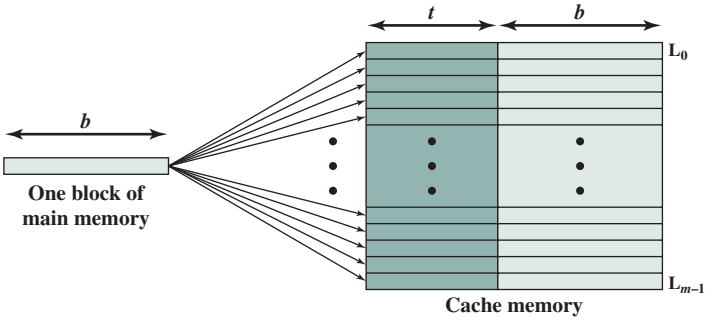
j = main memory block number

m = number of lines in the cache

Figure 4.8a shows the mapping for the first m blocks of main memory. Each block of main memory maps into one unique line of the cache. The next m blocks



(a) Direct mapping



(b) Associative mapping

Figure 4.8 Mapping from Main Memory to Cache: Direct and Associative

of main memory map into the cache in the same fashion; that is, block B_m of main memory maps into line L_0 of cache, block B_{m+1} maps into line L_1, and so on.

The mapping function is easily implemented using the main memory address. Figure 4.9 illustrates the general mechanism. For purposes of cache access, each main memory address can be viewed as consisting of three fields. The least significant w bits identify a unique word or byte within a block of main memory; in most contemporary machines, the address is at the byte level. The remaining s bits specify one of the 2^s blocks of main memory. The cache logic interprets these s bits as a tag of $s - r$ bits (most significant portion) and a line field of r bits. This latter field identifies one of the $m = 2^r$ lines of the cache. To summarize,

- Address length $= (s + w)$ bits
- Number of addressable units $= 2^{s+w}$ words or bytes
- Block size $=$ line size $= 2^w$ words or bytes
- Number of blocks in main memory $= \dfrac{2^{s+w}}{2^w} = 2^s$
- Number of lines in cache $= m = 2^r$
- Size of cache $= 2^{r+w}$ words or bytes
- Size of tag $= (s - r)$ bits

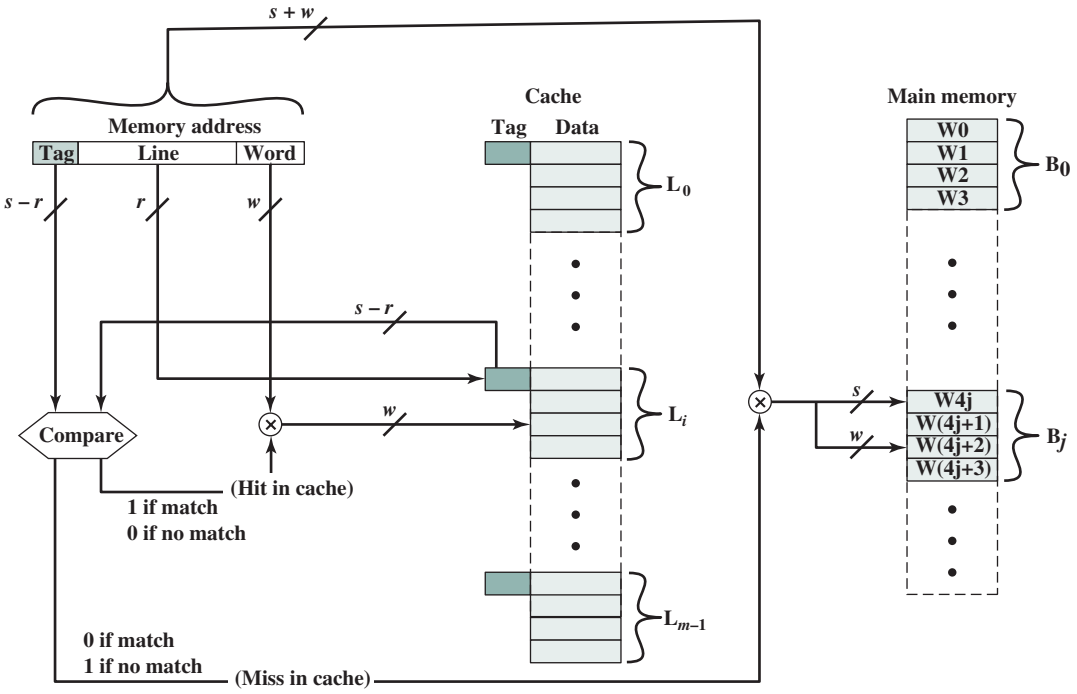


Figure 4.9 Direct-Mapping Cache Organization

EXAMPLE 4.2a Figure 4.10 shows our example system using direct mapping.[5] In the example, $m = 16\text{K} = 2^{14}$ and $i = j$ modulo 2^{14}. The mapping becomes

Cache Line	Starting Memory Address of Block
0	000000, 010000, …, FF0000
1	000004, 010004, …, FF0004
⋮	⋮
$2^{14} - 1$	00FFFC, 01FFFC, …, FFFFFC

Note that no two blocks that map into the same line number have the same tag number. Thus, blocks with starting addresses 000000, 010000, …, FF0000 have tag numbers 00, 01, …, FF, respectively.

Referring back to Figure 4.5, a read operation works as follows. The cache system is presented with a 24-bit address. The 14-bit line number is used as an index into the cache to access a particular line. If the 8-bit tag number matches the tag number currently stored in that line, then the 2-bit word number is used to select one of the 4 bytes in that line. Otherwise, the 22-bit tag-plus-line field is used to fetch a block from main memory. The actual address that is used for the fetch is the 22-bit tag-plus-line concatenated with two 0 bits, so that 4 bytes are fetched starting on a block boundary.

[5] In this and subsequent figures, memory values are represented in hexadecimal notation. See Chapter 9 for a basic refresher on number systems (decimal, binary, hexadecimal).

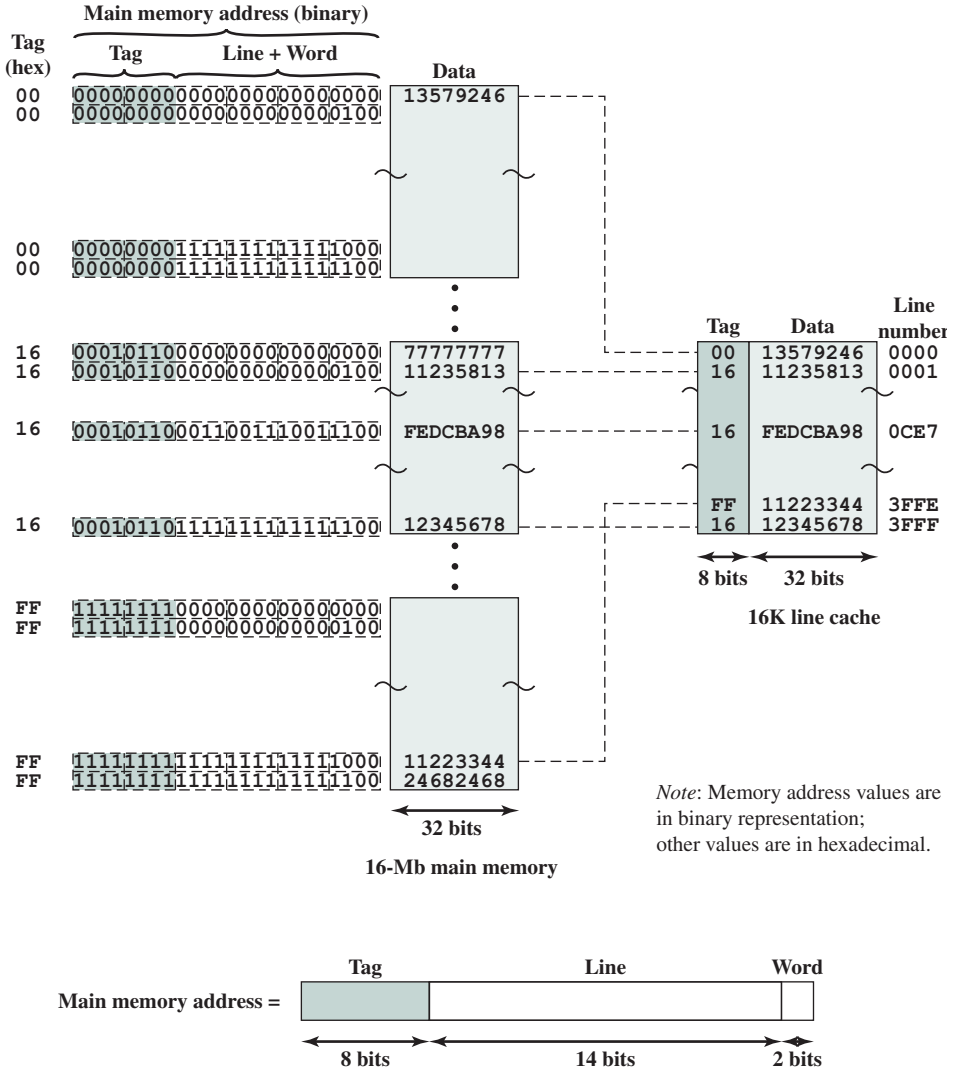


Figure 4.10 Direct Mapping Example

The effect of this mapping is that blocks of main memory are assigned to lines of the cache as follows:

Cache line	Main memory blocks assigned
0	$0, m, 2m, \ldots, 2^s - m$
1	$1, m + 1, 2m + 1, \ldots, 2^s - m + 1$
⋮	⋮
$m - 1$	$m - 1, 2m - 1, 3m - 1, \ldots, 2^s - 1$

Thus, the use of a portion of the address as a line number provides a unique mapping of each block of main memory into the cache. When a block is actually

read into its assigned line, it is necessary to tag the data to distinguish it from other blocks that can fit into that line. The most significant $s - r$ bits serve this purpose.

The direct mapping technique is simple and inexpensive to implement. Its main disadvantage is that there is a fixed cache location for any given block. Thus, if a program happens to reference words repeatedly from two different blocks that map into the same line, then the blocks will be continually swapped in the cache, and the hit ratio will be low (a phenomenon known as *thrashing*).

Selective Victim Cache Simulator

One approach to lower the **miss** penalty is to remember what was discarded in case it is needed again. Since the discarded data has already been fetched, it can be used again at a small cost. Such recycling is possible using a victim cache. Victim cache was originally proposed as an approach to reduce the conflict misses of direct mapped caches without affecting its fast access time. Victim cache is a fully associative cache, whose size is typically 4 to 16 cache lines, residing between a direct mapped L1 cache and the next level of memory. This concept is explored in Appendix F.

ASSOCIATIVE MAPPING Associative mapping overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of the cache (Figure 4.8b). In this case, the cache control logic interprets a memory address simply as a Tag and a Word field. The Tag field uniquely identifies a block of main memory. To determine whether a block is in the cache, the cache control logic must simultaneously examine every line's tag for a match. Figure 4.11 illustrates the logic.

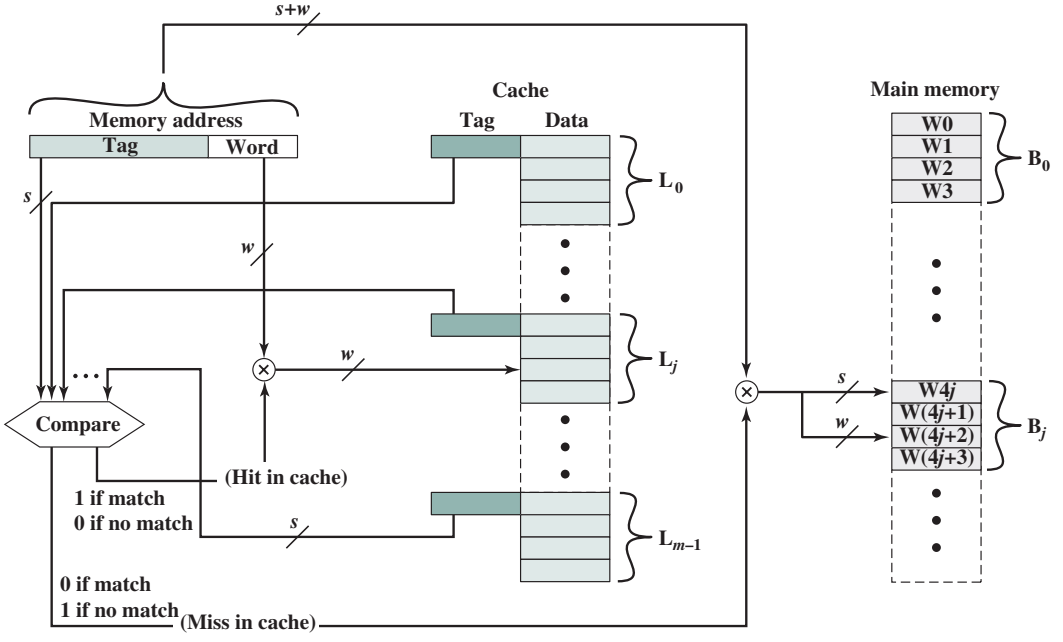


Figure 4.11 Fully Associative Cache Organization

EXAMPLE 4.2b Figure 4.12 shows our example using associative mapping. A main memory address consists of a 22-bit tag and a 2-bit byte number. The 22-bit tag must be stored with the 32-bit block of data for each line in the cache. Note that it is the leftmost (most significant) 22 bits of the address that form the tag. Thus, the 24-bit hexadecimal address 16339C has the 22-bit tag 058CE7. This is easily seen in binary notation:

Memory address	0001	0110	0011	0011	1001	1100	(binary)
	1	6	3	3	9	C	(hex)
Tag (leftmost 22 bits)	00	0101	1000	1100	1110	0111	(binary)
	0	5	8	C	E	7	(hex)

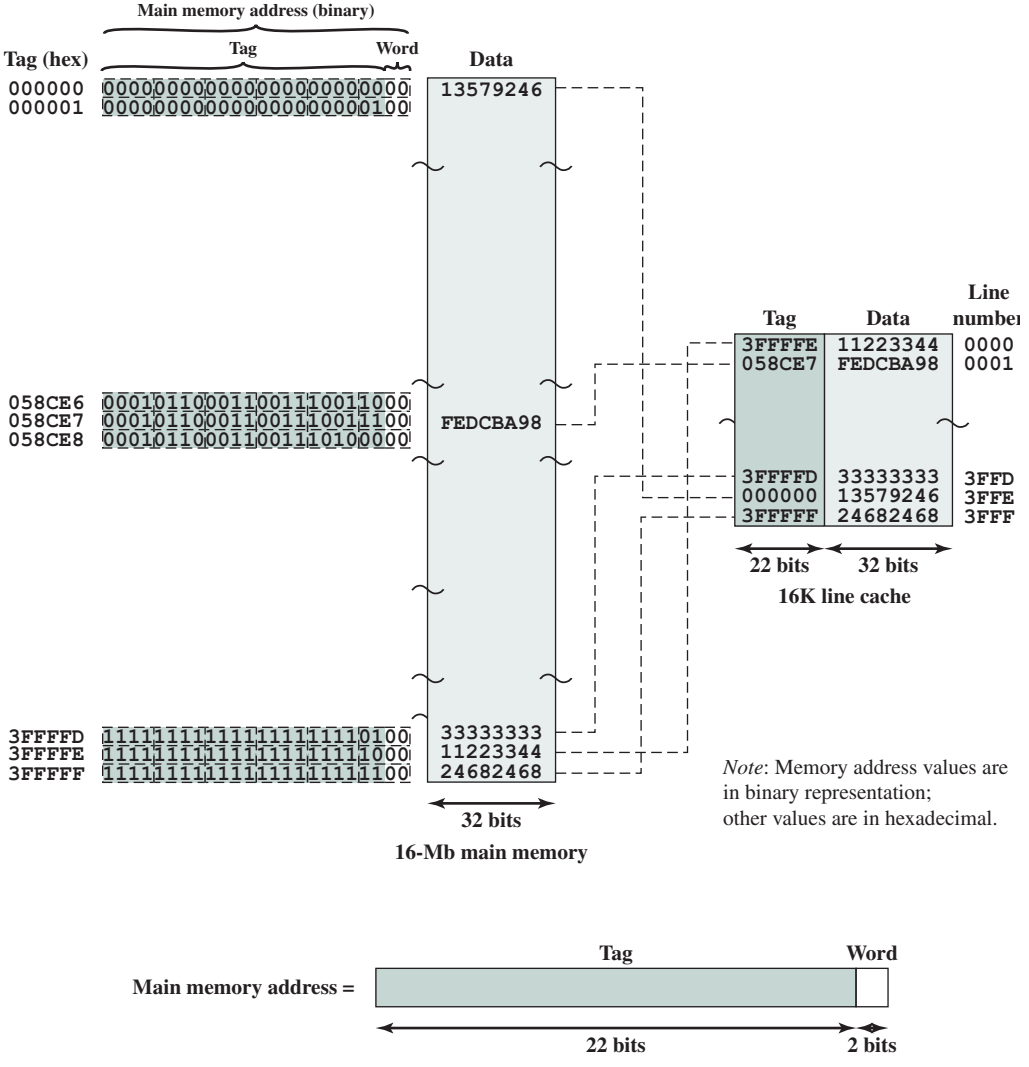


Figure 4.12 Associative Mapping Example

Note that no field in the address corresponds to the line number, so that the number of lines in the cache is not determined by the address format. To summarize,

- Address length $= (s + w)$ bits
- Number of addressable units $= 2^{s+w}$ words or bytes
- Block size = line size $= 2^w$ words or bytes
- Number of blocks in main memory $= \frac{2^{s+w}}{2^w} = 2^s$
- Number of lines in cache = undetermined
- Size of tag $= s$ bits

With associative mapping, there is flexibility as to which block to replace when a new block is read into the cache. Replacement algorithms, discussed later in this section, are designed to maximize the hit ratio. The principal disadvantage of associative mapping is the complex circuitry required to examine the tags of all cache lines in parallel.

Cache Time Analysis Simulator

SET-ASSOCIATIVE MAPPING Set-associative mapping is a compromise that exhibits the strengths of both the direct and associative approaches while reducing their disadvantages.

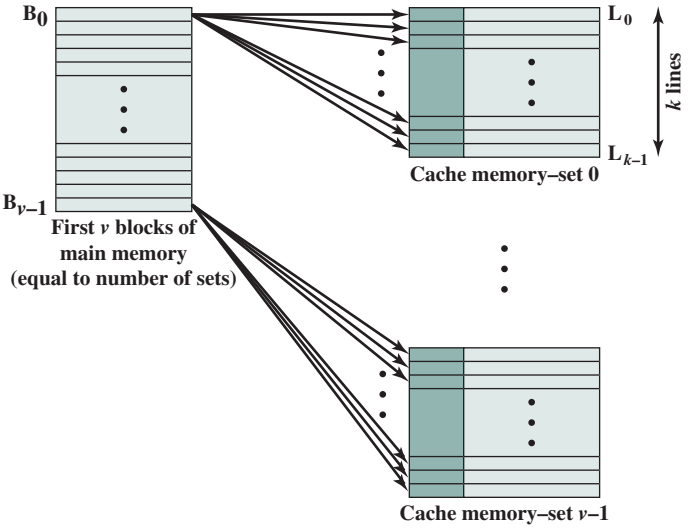
In this case, the cache consists of number sets, each of which consists of a number of lines. The relationships are

$$m = v \times k$$
$$i = j \text{ modulo } v$$

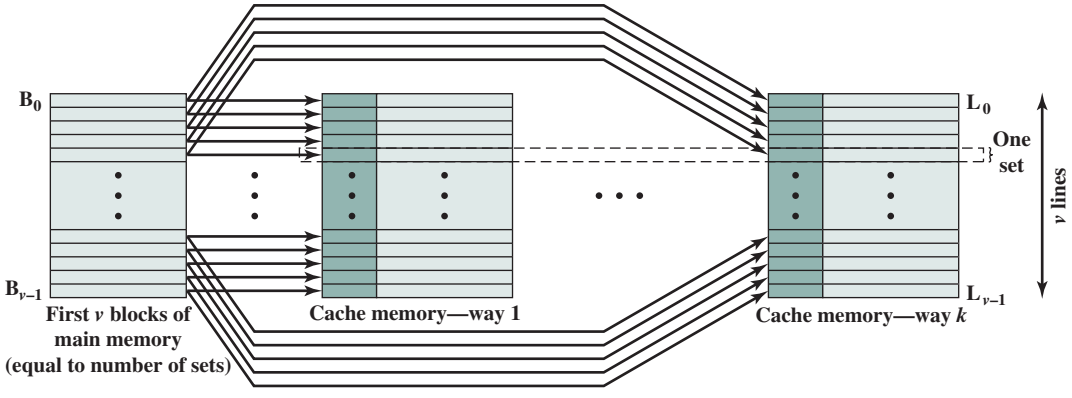
where

i = cache set number
j = main memory block number
m = number of lines in the cache
v = number of sets
k = number of lines in each set

This is referred to as k-way set-associative mapping. With set-associative mapping, block B_j can be mapped into any of the lines of set j. Figure 4.13a illustrates this mapping for the first v blocks of main memory. As with associative mapping, each word maps into multiple cache lines. For set-associative mapping, each word maps into all the cache lines in a specific set, so that main memory block B_0 maps into set 0, and so on. Thus, the set-associative cache can be physically implemented as v associative caches. It is also possible to implement the set-associative cache as k direct mapping caches, as shown in Figure 4.13b. Each direct-mapped cache is referred to as a *way,* consisting of v lines. The first v lines of main memory are direct mapped into the v lines of each way; the next group of v lines of main memory are similarly mapped, and so on. The direct-mapped implementation is typically used



(a) *v* associative–mapped caches



(b) *k* direct–mapped caches

Figure 4.13 Mapping from Main Memory to Cache: *k*-Way Set Associative

for small degrees of associativity (small values of k) while the associative-mapped implementation is typically used for higher degrees of associativity [JACO08].

For set-associative mapping, the cache control logic interprets a memory address as three fields: Tag, Set, and Word. The d set bits specify one of $v = 2^d$ sets. The s bits of the Tag and Set fields specify one of the 2^s blocks of main memory. Figure 4.14 illustrates the cache control logic. With fully associative mapping, the tag in a memory address is quite large and must be compared to the tag of every line in the cache. With k-way set-associative mapping, the tag in a memory address is much smaller and is only compared to the k tags within a single set. To summarize,

- Address length $= (s + w)$ bits
- Number of addressable units $= 2^{s+w}$ words or bytes

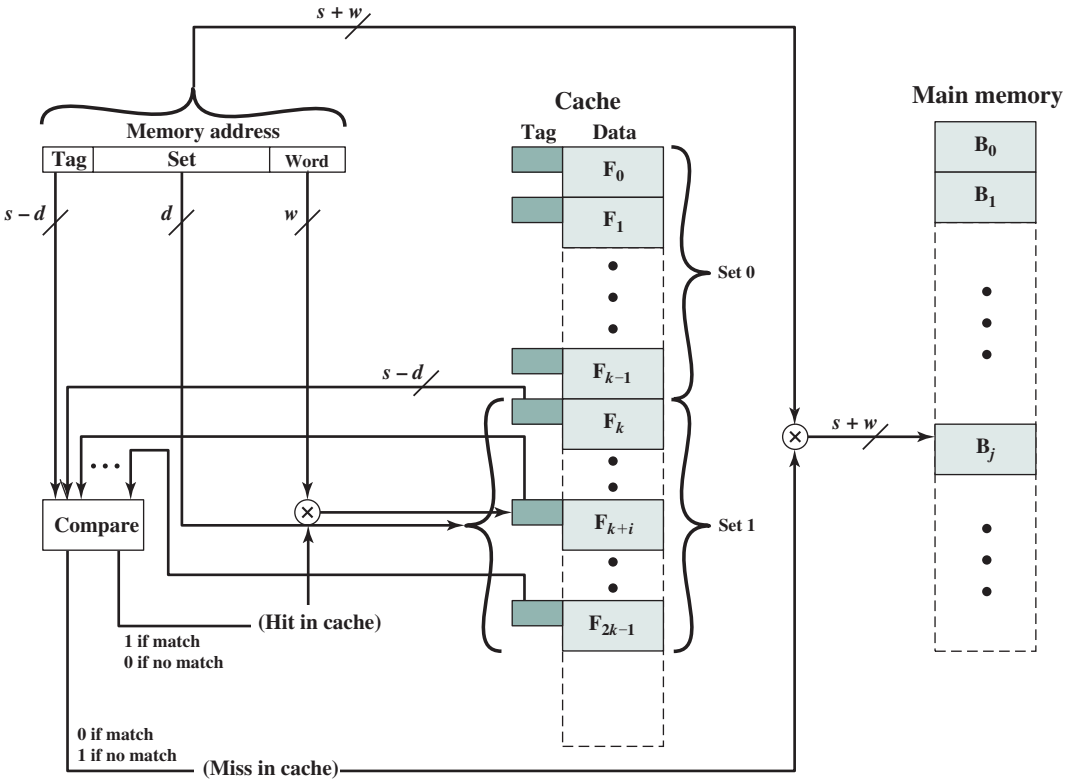


Figure 4.14 *k*-Way Set-Associative Cache Organization

- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = $\frac{2^{s+w}}{2^w} = 2^s$
- Number of lines in set = k
- Number of sets = $v = 2^d$
- Number of lines in cache = $m = kv = k \times 2^d$
- Size of cache = $k \times 2^{d+w}$ words or bytes
- Size of tag = $(s - d)$ bits

EXAMPLE 4.2c Figure 4.15 shows our example using set-associative mapping with two lines in each set, referred to as two-way set-associative. The 13-bit set number identifies a unique set of two lines within the cache. It also gives the number of the block in main memory, modulo 2^{13}. This determines the mapping of blocks into lines. Thus, blocks 000000, 008000, …, FF8000 of main memory map into cache set 0. Any of those blocks can be loaded into either of the two lines in the set. Note that no two blocks that map into the same cache set have the same tag number. For a read operation, the 13-bit set number is used to determine which set of two lines is to be examined. Both lines in the set are examined for a match with the tag number of the address to be accessed.

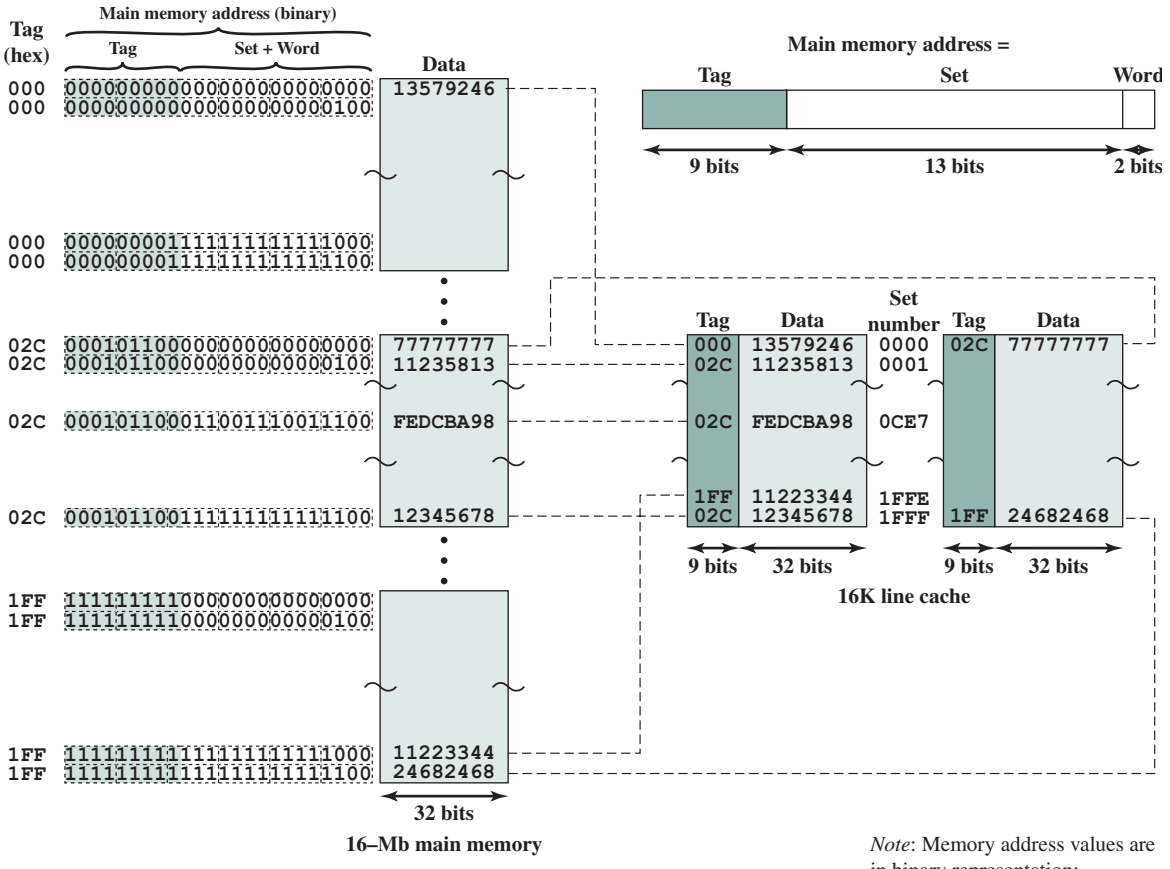


Figure 4.15 Two-Way Set-Associative Mapping Example

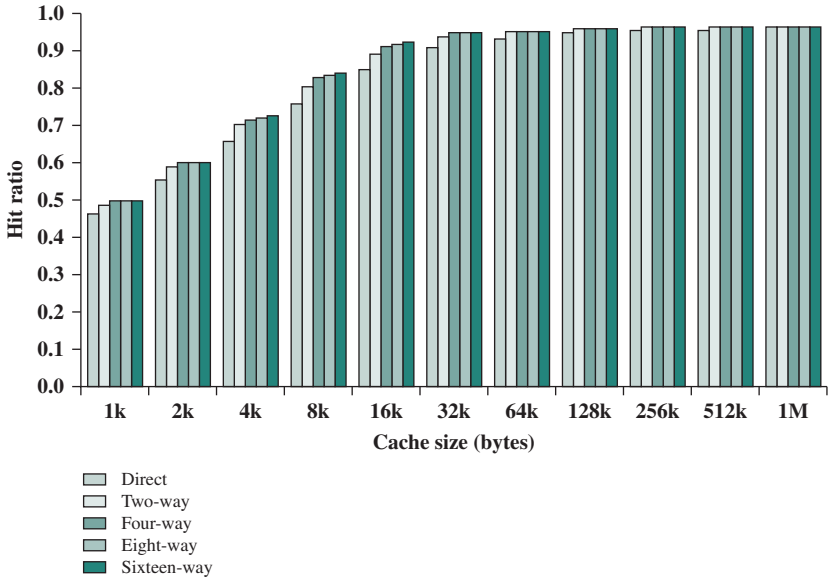


Figure 4.16 Varying Associativity over Cache Size

In the extreme case of $v = m$, $k = 1$, the set-associative technique reduces to direct mapping, and for $v = 1$, $k = m$, it reduces to associative mapping. The use of two lines per set ($v = m/2$, $k = 2$) is the most common set-associative organization. It significantly improves the hit ratio over direct mapping. Four-way set associative ($v = m/4$, $k = 4$) makes a modest additional improvement for a relatively small additional cost [MAYB84, HILL89]. Further increases in the number of lines per set have little effect.

Figure 4.16 shows the results of one simulation study of set-associative cache performance as a function of cache size [GENU04]. The difference in performance between direct and two-way set associative is significant up to at least a cache size of 64 kB. Note also that the difference between two-way and four-way at 4 kB is much less than the difference in going from for 4 kB to 8 kB in cache size. The complexity of the cache increases in proportion to the associativity, and in this case would not be justifiable against increasing cache size to 8 or even 16 kB. A final point to note is that beyond about 32 kB, increase in cache size brings no significant increase in performance.

The results of Figure 4.16 are based on simulating the execution of a GCC compiler. Different applications may yield different results. For example, [CANT01] reports on the results for cache performance using many of the CPU2000 SPEC benchmarks. The results of [CANT01] in comparing hit ratio to cache size follow the same pattern as Figure 4.16, but the specific values are somewhat different.

Cache Simulator
Multitask Cache Simulator

Replacement Algorithms

Once the cache has been filled, when a new block is brought into the cache, one of the existing blocks must be replaced. For direct mapping, there is only one possible line for any particular block, and no choice is possible. For the associative and set-associative techniques, a replacement algorithm is needed. To achieve high speed, such an algorithm must be implemented in hardware. A number of algorithms have been tried. We mention four of the most common. Probably the most effective is **least recently used (LRU)**: Replace that block in the set that has been in the cache longest with no reference to it. For two-way set associative, this is easily implemented. Each line includes a USE bit. When a line is referenced, its USE bit is set to 1 and the USE bit of the other line in that set is set to 0. When a block is to be read into the set, the line whose USE bit is 0 is used. Because we are assuming that more recently used memory locations are more likely to be referenced, LRU should give the best hit ratio. LRU is also relatively easy to implement for a fully associative cache. The cache mechanism maintains a separate list of indexes to all the lines in the cache. When a line is referenced, it moves to the front of the list. For replacement, the line at the back of the list is used. Because of its simplicity of implementation, LRU is the most popular replacement algorithm.

Another possibility is first-in-first-out (FIFO): Replace that block in the set that has been in the cache longest. FIFO is easily implemented as a round-robin or circular buffer technique. Still another possibility is least frequently used (LFU): Replace that block in the set that has experienced the fewest references. LFU could be implemented by associating a counter with each line. A technique not based on usage (i.e., not LRU, LFU, FIFO, or some variant) is to pick a line at random from among the candidate lines. Simulation studies have shown that random replacement provides only slightly inferior performance to an algorithm based on usage [SMIT82].

Write Policy

When a block that is resident in the cache is to be replaced, there are two cases to consider. If the old block in the cache has not been altered, then it may be overwritten with a new block without first writing out the old block. If at least one write operation has been performed on a word in that line of the cache, then main memory must be updated by writing the line of cache out to the block of memory before bringing in the new block. A variety of write policies, with performance and economic trade-offs, is possible. There are two problems to contend with. First, more than one device may have access to main memory. For example, an I/O module may be able to read-write directly to memory. If a word has been altered only in the cache, then the corresponding memory word is invalid. Further, if the I/O device has altered main memory, then the cache word is invalid. A more complex problem occurs when multiple processors are attached to the same bus and each processor has its own local cache. Then, if a word is altered in one cache, it could conceivably invalidate a word in other caches.

The simplest technique is called **write through**. Using this technique, all write operations are made to main memory as well as to the cache, ensuring that main memory is always valid. Any other processor–cache module can monitor traffic to main memory to maintain consistency within its own cache. The main disadvantage

of this technique is that it generates substantial memory traffic and may create a bottleneck. An alternative technique, known as **write back**, minimizes memory writes. With write back, updates are made only in the cache. When an update occurs, a **dirty bit**, or **use bit**, associated with the line is set. Then, when a block is replaced, it is written back to main memory if and only if the dirty bit is set. The problem with write back is that portions of main memory are invalid, and hence accesses by I/O modules can be allowed only through the cache. This makes for complex circuitry and a potential bottleneck. Experience has shown that the percentage of memory references that are writes is on the order of 15% [SMIT82]. However, for HPC applications, this number may approach 33% (vector-vector multiplication) and can go as high as 50% (matrix transposition).

EXAMPLE 4.3 Consider a cache with a line size of 32 bytes and a main memory that requires 30 ns to transfer a 4-byte word. For any line that is written at least once before being swapped out of the cache, what is the average number of times that the line must be written before being swapped out for a write-back cache to be more efficient that a write-through cache?

For the write-back case, each dirty line is written back once, at swap-out time, taking $8 \times 30 = 240$ ns. For the write-through case, each update of the line requires that one word be written out to main memory, taking 30 ns. Therefore, if the average line that gets written at least once gets written more than 8 times before swap out, then write back is more efficient.

In a bus organization in which more than one device (typically a processor) has a cache and main memory is shared, a new problem is introduced. If data in one cache are altered, this invalidates not only the corresponding word in main memory, but also that same word in other caches (if any other cache happens to have that same word). Even if a write-through policy is used, the other caches may contain invalid data. A system that prevents this problem is said to maintain cache coherency. Possible approaches to cache coherency include the following:

- **Bus watching with write through:** Each cache controller monitors the address lines to detect write operations to memory by other bus masters. If another master writes to a location in shared memory that also resides in the cache memory, the cache controller invalidates that cache entry. This strategy depends on the use of a write-through policy by all cache controllers.
- **Hardware transparency:** Additional hardware is used to ensure that all updates to main memory via cache are reflected in all caches. Thus, if one processor modifies a word in its cache, this update is written to main memory. In addition, any matching words in other caches are similarly updated.
- **Noncacheable memory:** Only a portion of main memory is shared by more than one processor, and this is designated as noncacheable. In such a system, all accesses to shared memory are cache misses, because the shared memory is never copied into the cache. The noncacheable memory can be identified using chip-select logic or high-address bits.

Cache coherency is an active field of research. This topic is explored further in Part Five.

Line Size

Another design element is the line size. When a block of data is retrieved and placed in the cache, not only the desired word but also some number of adjacent words are retrieved. As the block size increases from very small to larger sizes, the hit ratio will at first increase because of the principle of locality, which states that data in the vicinity of a referenced word are likely to be referenced in the near future. As the block size increases, more useful data are brought into the cache. The hit ratio will begin to decrease, however, as the block becomes even bigger and the probability of using the newly fetched information becomes less than the probability of reusing the information that has to be replaced. Two specific effects come into play:

- Larger blocks reduce the number of blocks that fit into a cache. Because each block fetch overwrites older cache contents, a small number of blocks results in data being overwritten shortly after they are fetched.
- As a block becomes larger, each additional word is farther from the requested word and therefore less likely to be needed in the near future.

The relationship between block size and hit ratio is complex, depending on the locality characteristics of a particular program, and no definitive optimum value has been found. A size of from 8 to 64 bytes seems reasonably close to optimum [SMIT87, PRZY88, PRZY90, HAND98]. For HPC systems, 64- and 128-byte cache line sizes are most frequently used.

Number of Caches

When caches were originally introduced, the typical system had a single cache. More recently, the use of multiple caches has become the norm. Two aspects of this design issue concern the number of levels of caches and the use of unified versus split caches.

MULTILEVEL CACHES As logic density has increased, it has become possible to have a cache on the same chip as the processor: the on-chip cache. Compared with a cache reachable via an external bus, the on-chip cache reduces the processor's external bus activity and therefore speeds up execution times and increases overall system performance. When the requested instruction or data is found in the on-chip cache, the bus access is eliminated. Because of the short data paths internal to the processor, compared with bus lengths, on-chip cache accesses will complete appreciably faster than would even zero-wait state bus cycles. Furthermore, during this period the bus is free to support other transfers.

The inclusion of an on-chip cache leaves open the question of whether an off-chip, or external, cache is still desirable. Typically, the answer is yes, and most contemporary designs include both on-chip and external caches. The simplest such organization is known as a two-level cache, with the internal level 1 (L1) and the external cache designated as level 2 (L2). The reason for including an L2 cache is the following: If there is no L2 cache and the processor makes an access request for a memory location not in the L1 cache, then the processor must access DRAM or

ROM memory across the bus. Due to the typically slow bus speed and slow memory access time, this results in poor performance. On the other hand, if an L2 SRAM (static RAM) cache is used, then frequently the missing information can be quickly retrieved. If the SRAM is fast enough to match the bus speed, then the data can be accessed using a zero-wait state transaction, the fastest type of bus transfer.

Two features of contemporary cache design for multilevel caches are noteworthy. First, for an off-chip L2 cache, many designs do not use the system bus as the path for transfer between the L2 cache and the processor, but use a separate data path, so as to reduce the burden on the system bus. Second, with the continued shrinkage of processor components, a number of processors now incorporate the L2 cache on the processor chip, improving performance.

The potential savings due to the use of an L2 cache depends on the hit rates in both the L1 and L2 caches. Several studies have shown that, in general, the use of a second-level cache does improve performance (e.g., see [AZIM92], [NOVI93], [HAND98]). However, the use of multilevel caches does complicate all of the design issues related to caches, including size, replacement algorithm, and write policy; see [HAND98] and [PEIR99] for discussions.

Figure 4.17 shows the results of one simulation study of two-level cache performance as a function of cache size [GENU04]. The figure assumes that both caches have the same line size and shows the total hit ratio. That is, a hit is counted if the desired data appears in either the L1 or the L2 cache. The figure shows the impact of L2 on total hits with respect to L1 size. L2 has little effect on the total number of cache hits until it is at least double the L1 cache size. Note that the steepest part of the slope for an L1 cache of 8 kB is for an L2 cache of 16 kB. Again for an L1 cache of 16 kB, the steepest part of the curve is for an L2 cache size of 32 kB. Prior to that point, the L2 cache has little, if any, impact on total cache performance. The need for the L2 cache to be larger than

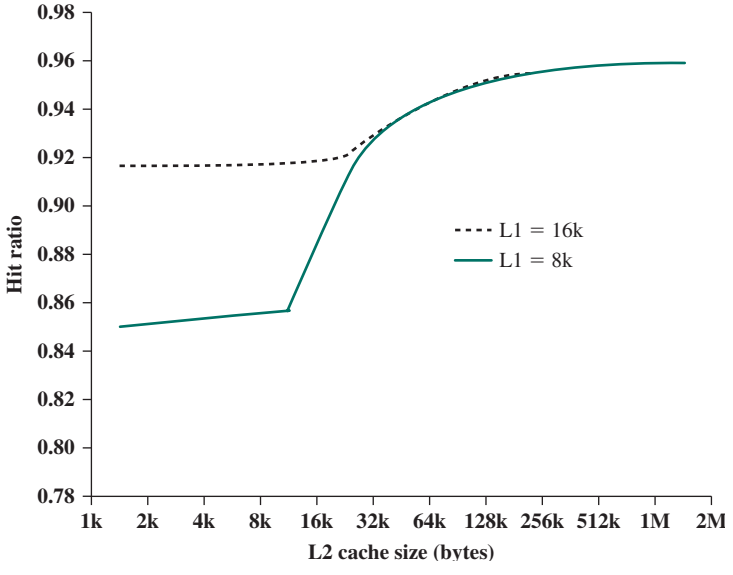


Figure 4.17 Total Hit Ratio (L1 and L2) for 8-kB and 16-kB L1

the L1 cache to affect performance makes sense. If the L2 cache has the same line size and capacity as the L1 cache, its contents will more or less mirror those of the L1 cache.

With the increasing availability of on-chip area available for cache, most contemporary microprocessors have moved the L2 cache onto the processor chip and added an L3 cache. Originally, the L3 cache was accessible over the external bus. More recently, most microprocessors have incorporated an on-chip L3 cache. In either case, there appears to be a performance advantage to adding the third level (e.g., see [GHAI98]). Further, large systems, such as the IBM mainframe zEnterprise systems, now incorporate 3 on-chip cache levels and a fourth level of cache shared across multiple chips [CURR11].

UNIFIED VERSUS SPLIT CACHES When the on-chip cache first made an appearance, many of the designs consisted of a single cache used to store references to both data and instructions. More recently, it has become common to split the cache into two: one dedicated to instructions and one dedicated to data. These two caches both exist at the same level, typically as two L1 caches. When the processor attempts to fetch an instruction from main memory, it first consults the instruction L1 cache, and when the processor attempts to fetch data from main memory, it first consults the data L1 cache.

There are two potential advantages of a unified cache:

- For a given cache size, a unified cache has a higher hit rate than split caches because it balances the load between instruction and data fetches automatically. That is, if an execution pattern involves many more instruction fetches than data fetches, then the cache will tend to fill up with instructions, and if an execution pattern involves relatively more data fetches, the opposite will occur.
- Only one cache needs to be designed and implemented.

The trend is toward split caches at the L1 and unified caches for higher levels, particularly for superscalar machines, which emphasize parallel instruction execution and the prefetching of predicted future instructions. The key advantage of the split cache design is that it eliminates contention for the cache between the instruction fetch/decode unit and the execution unit. This is important in any design that relies on the pipelining of instructions. Typically, the processor will fetch instructions ahead of time and fill a buffer, or pipeline, with instructions to be executed. Suppose now that we have a unified instruction/data cache. When the execution unit performs a memory access to load and store data, the request is submitted to the unified cache. If, at the same time, the instruction prefetcher issues a read request to the cache for an instruction, that request will be temporarily blocked so that the cache can service the execution unit first, enabling it to complete the currently executing instruction. This cache contention can degrade performance by interfering with efficient use of the instruction pipeline. The split cache structure overcomes this difficulty.

4.4 PENTIUM 4 CACHE ORGANIZATION

The evolution of cache organization is seen clearly in the evolution of Intel microprocessors (Table 4.4). The 80386 does not include an on-chip cache. The 80486 includes a single on-chip cache of 8 kB, using a line size of 16 bytes and a four-way

Table 4.4 Intel Cache Evolution

Problem	Solution	Processor on Which Feature First Appears
External memory slower than the system bus.	Add external cache using faster memory technology.	386
Increased processor speed results in external bus becoming a bottleneck for cache access.	Move external cache on-chip, operating at the same speed as the processor.	486
Internal cache is rather small, due to limited space on chip.	Add external L2 cache using faster technology than main memory.	486
Contention occurs when both the Instruction Prefetcher and the Execution Unit simultaneously require access to the cache. In that case, the Prefetcher is stalled while the Execution Unit's data access takes place.	Create separate data and instruction caches.	Pentium
Increased processor speed results in external bus becoming a bottleneck for L2 cache access.	Create separate back-side bus that runs at higher speed than the main (front-side) external bus. The BSB is dedicated to the L2 cache.	Pentium Pro
	Move L2 cache on to the processor chip.	Pentium II
Some applications deal with massive databases and must have rapid access to large amounts of data. The on-chip caches are too small.	Add external L3 cache.	Pentium III
	Move L3 cache on-chip.	Pentium 4

set-associative organization. All of the Pentium processors include two on-chip L1 caches, one for data and one for instructions. For the Pentium 4, the L1 data cache is 16 kB, using a line size of 64 bytes and a four-way set-associative organization. The Pentium 4 instruction cache is described subsequently. The Pentium II also includes an L2 cache that feeds both of the L1 caches. The L2 cache is eight-way set associative with a size of 512 kB and a line size of 128 bytes. An L3 cache was added for the Pentium III and became on-chip with high-end versions of the Pentium 4.

Figure 4.18 provides a simplified view of the Pentium 4 organization, highlighting the placement of the three caches. The processor core consists of four major components:

- **Fetch/decode unit:** Fetches program instructions in order from the L2 cache, decodes these into a series of micro-operations, and stores the results in the L1 instruction cache.
- **Out-of-order execution logic:** Schedules execution of the micro-operations subject to data dependencies and resource availability; thus, micro-operations may be scheduled for execution in a different order than they were fetched from the instruction stream. As time permits, this unit schedules speculative execution of micro-operations that may be required in the future.

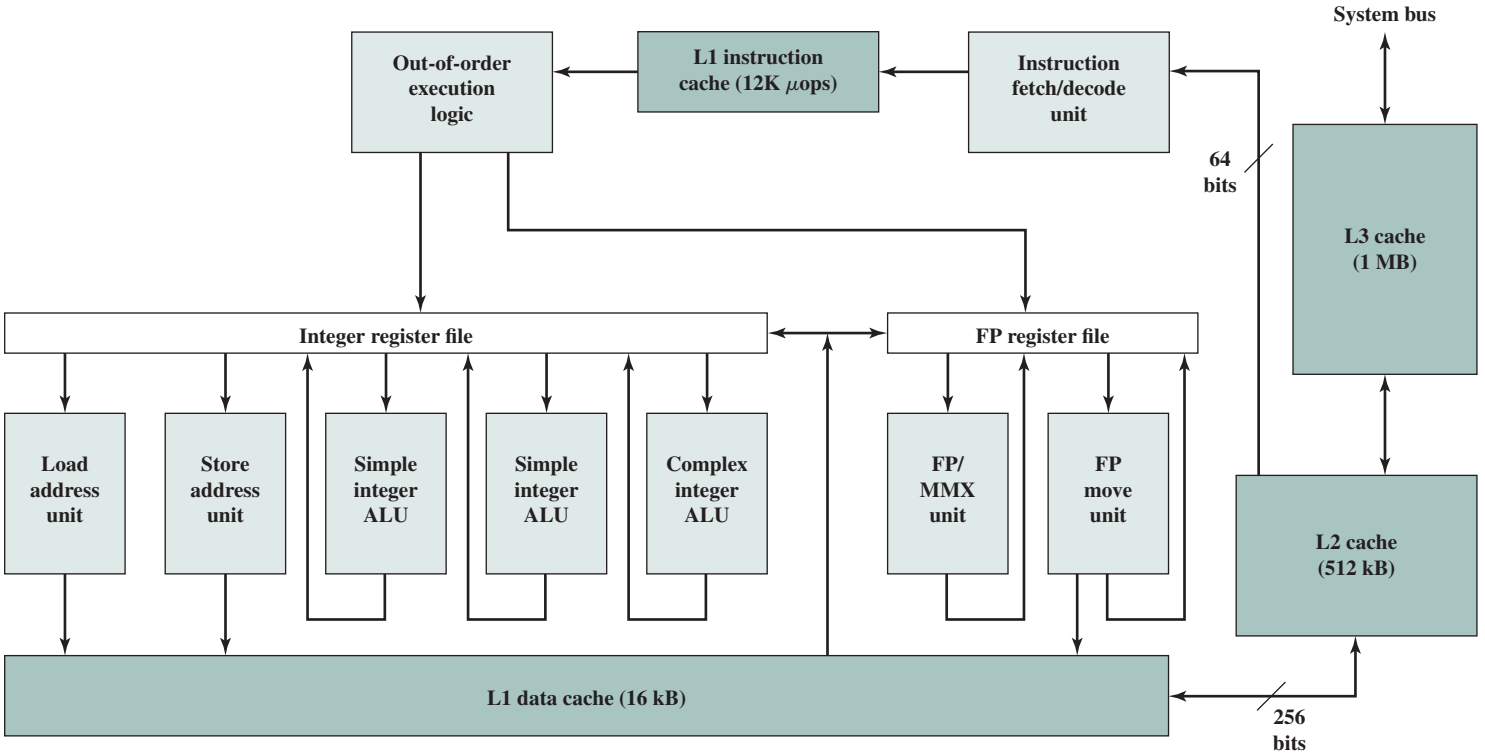


Figure 4.18 Pentium 4 Block Diagram

Table 4.5 Pentium 4 Cache Operating Modes

Control Bits		Operating Mode		
CD	NW	Cache Fills	Write Throughs	Invalidates
0	0	Enabled	Enabled	Enabled
1	0	Disabled	Enabled	Enabled
1	1	Disabled	Disabled	Disabled

Note: CD = 0; NW = 1 is an invalid combination.

- **Execution units:** These units execute micro-operations, fetching the required data from the L1 data cache and temporarily storing results in registers.
- **Memory subsystem:** This unit includes the L2 and L3 caches and the system bus, which is used to access main memory when the L1 and L2 caches have a cache miss and to access the system I/O resources.

Unlike the organization used in all previous Pentium models, and in most other processors, the Pentium 4 instruction cache sits between the instruction decode logic and the execution core. The reasoning behind this design decision is as follows: As discussed more fully in Chapter 16, the Pentium process decodes, or translates, Pentium machine instructions into simple RISC-like instructions called micro-operations. The use of simple, fixed-length micro-operations enables the use of superscalar pipelining and scheduling techniques that enhance performance. However, the Pentium machine instructions are cumbersome to decode; they have a variable number of bytes and many different options. It turns out that performance is enhanced if this decoding is done independently of the scheduling and pipelining logic. We return to this topic in Chapter 16.

The data cache employs a write-back policy: Data are written to main memory only when they are removed from the cache and there has been an update. The Pentium 4 processor can be dynamically configured to support write-through caching.

The L1 data cache is controlled by two bits in one of the control registers, labeled the CD (cache disable) and NW (not write-through) bits (Table 4.5). There are also two Pentium 4 instructions that can be used to control the data cache: INVD invalidates (flushes) the internal cache memory and signals the external cache (if any) to invalidate. WBINVD writes back and invalidates internal cache and then writes back and invalidates external cache.

Both the L2 and L3 caches are eight-way set-associative with a line size of 128 bytes.

4.5 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

access time	cache line	cache set
associative mapping	cache memory	data cache
cache hit	cache miss	direct access

direct mapping	logical cache	spatial locality
high-performance computing (HPC)	memory hierarchy	split cache
hit	miss	tag
hit ratio	multilevel cache	temporal locality
instruction cache	physical address	unified cache
L1 cache	physical cache	virtual address
L2 cache	random access	virtual cache
L3 cache	replacement algorithm	write back
line	secondary memory	write through
locality	sequential access	
	set-associative mapping	

Review Questions

4.1 What are the differences among sequential access, direct access, and random access?

4.2 What is the general relationship among access time, memory cost, and capacity?

4.3 How does the principle of locality relate to the use of multiple memory levels?

4.4 What are the differences among direct mapping, associative mapping, and set-associative mapping?

4.5 For a direct-mapped cache, a main memory address is viewed as consisting of three fields. List and define the three fields.

4.6 For an associative cache, a main memory address is viewed as consisting of two fields. List and define the two fields.

4.7 For a set-associative cache, a main memory address is viewed as consisting of three fields. List and define the three fields.

4.8 What is the distinction between spatial locality and temporal locality?

4.9 In general, what are the strategies for exploiting spatial locality and temporal locality?

Problems

4.1 A set-associative cache consists of 64 lines, or slots, divided into four-line sets. Main memory contains 4K blocks of 128 words each. Show the format of main memory addresses.

4.2 A two-way set-associative cache has lines of 16 bytes and a total size of 8 kB. The 64-MB main memory is byte addressable. Show the format of main memory addresses.

4.3 For the hexadecimal main memory addresses 111111, 666666, BBBBBB, show the following information, in hexadecimal format:

- **a.** Tag, Line, and Word values for a direct-mapped cache, using the format of Figure 4.10
- **b.** Tag and Word values for an associative cache, using the format of Figure 4.12
- **c.** Tag, Set, and Word values for a two-way set-associative cache, using the format of Figure 4.15

4.4 List the following values:

- **a.** For the direct cache example of Figure 4.10: address length, number of addressable units, block size, number of blocks in main memory, number of lines in cache, size of tag
- **b.** For the associative cache example of Figure 4.12: address length, number of addressable units, block size, number of blocks in main memory, number of lines in cache, size of tag

c. For the two-way set-associative cache example of Figure 4.15: address length, number of addressable units, block size, number of blocks in main memory, number of lines in set, number of sets, number of lines in cache, size of tag

4.5 Consider a 32-bit microprocessor that has an on-chip 16-kB four-way set-associative cache. Assume that the cache has a line size of four 32-bit words. Draw a block diagram of this cache showing its organization and how the different address fields are used to determine a cache hit/miss. Where in the cache is the word from memory location ABCDE8F8 mapped?

4.6 Given the following specifications for an external cache memory: four-way set associative; line size of two 16-bit words; able to accommodate a total of 4K 32-bit words from main memory; used with a 16-bit processor that issues 24-bit addresses. Design the cache structure with all pertinent information and show how it interprets the processor's addresses.

4.7 The Intel 80486 has an on-chip, unified cache. It contains 8 kB and has a four-way set-associative organization and a block length of four 32-bit words. The cache is organized into 128 sets. There is a single "line valid bit" and three bits, B0, B1, and B2 (the "LRU" bits), per line. On a cache miss, the 80486 reads a 16-byte line from main memory in a bus memory read burst. Draw a simplified diagram of the cache and show how the different fields of the address are interpreted.

4.8 Consider a machine with a byte addressable main memory of 2^{16} bytes and block size of 8 bytes. Assume that a direct mapped cache consisting of 32 lines is used with this machine.

a. How is a 16-bit memory address divided into tag, line number, and byte number?

b. Into what line would bytes with each of the following addresses be stored?

0001	0001	0001	1011
1100	0011	0011	0100
1101	0000	0001	1101
1010	1010	1010	1010

c. Suppose the byte with address 0001 1010 0001 1010 is stored in the cache. What are the addresses of the other bytes stored along with it?

d. How many total bytes of memory can be stored in the cache?

e. Why is the tag also stored in the cache?

4.9 For its on-chip cache, the Intel 80486 uses a replacement algorithm referred to as **pseudo least recently used**. Associated with each of the 128 sets of four lines (labeled L0, L1, L2, L3) are three bits B0, B1, and B2. The replacement algorithm works as follows: When a line must be replaced, the cache will first determine whether the most recent use was from L0 and L1 or L2 and L3. Then the cache will determine which of the pair of blocks was least recently used and mark it for replacement. Figure 4.19 illustrates the logic.

a. Specify how the bits B0, B1, and B2 are set and then describe in words how they are used in the replacement algorithm depicted in Figure 4.19.

b. Show that the 80486 algorithm approximates a true LRU algorithm. *Hint:* Consider the case in which the most recent order of usage is L0, L2, L3, L1.

c. Demonstrate that a true LRU algorithm would require 6 bits per set.

4.10 A set-associative cache has a block size of four 16-bit words and a set size of 2. The cache can accommodate a total of 4096 words. The main memory size that is cacheable is 64K $\times$ 32 bits. Design the cache structure and show how the processor's addresses are interpreted.

4.11 Consider a memory system that uses a 32-bit address to address at the byte level, plus a cache that uses a 64-byte line size.

a. Assume a direct mapped cache with a tag field in the address of 20 bits. Show the address format and determine the following parameters: number of addressable units, number of blocks in main memory, number of lines in cache, size of tag.

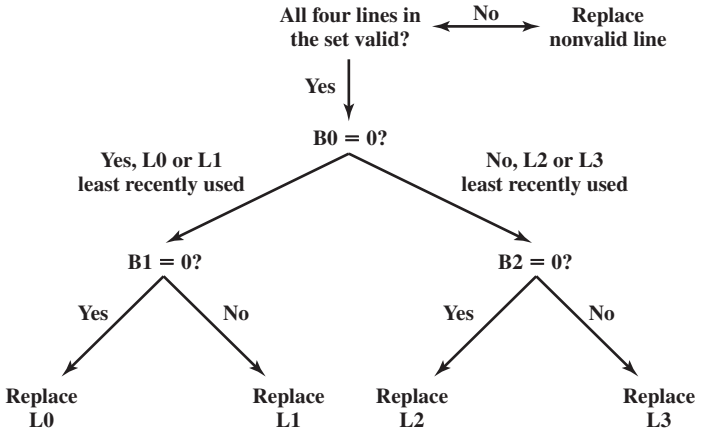


Figure 4.19 Intel 80486 On-Chip Cache Replacement Strategy

b. Assume an associative cache. Show the address format and determine the following parameters: number of addressable units, number of blocks in main memory, number of lines in cache, size of tag.

c. Assume a four-way set-associative cache with a tag field in the address of 9 bits. Show the address format and determine the following parameters: number of addressable units, number of blocks in main memory, number of lines in set, number of sets in cache, number of lines in cache, size of tag.

4.12 Consider a computer with the following characteristics: total of 1 MB of main memory; word size of 1 byte; block size of 16 bytes; and cache size of 64 kB.

a. For the main memory addresses of F0010, 01234, and CABBE, give the corresponding tag, cache line address, and word offsets for a direct-mapped cache.

b. Give any two main memory addresses with different tags that map to the same cache slot for a direct-mapped cache.

c. For the main memory addresses of F0010 and CABBE, give the corresponding tag and offset values for a fully-associative cache.

d. For the main memory addresses of F0010 and CABBE, give the corresponding tag, cache set, and offset values for a two-way set-associative cache.

4.13 Describe a simple technique for implementing an LRU replacement algorithm in a four-way set-associative cache.

4.14 Consider again Example 4.3. How does the answer change if the main memory uses a block transfer capability that has a first-word access time of 30 ns and an access time of 5 ns for each word thereafter?

4.15 Consider the following code:

```
for (i = 0; i < 20; i++)
    for (j = 0; j < 10; j++)
        a[i] = a[i]*j
```

a. Give one example of the spatial locality in the code.

b. Give one example of the temporal locality in the code.

4.16 Generalize Equations (4.2) and (4.3), in Appendix 4A, to *N*-level memory hierarchies.

4.17 A computer system contains a main memory of 32K 16-bit words. It also has a 4K word cache divided into four-line sets with 64 words per line. Assume that the cache is initially empty. The processor fetches words from locations 0, 1, 2, . . . , 4351 in that

order. It then repeats this fetch sequence nine more times. The cache is 10 times faster than main memory. Estimate the improvement resulting from the use of the cache. Assume an LRU policy for block replacement.

4.18 Consider a cache of 4 lines of 16 bytes each. Main memory is divided into blocks of 16 bytes each. That is, block 0 has bytes with addresses 0 through 15, and so on. Now consider a program that accesses memory in the following sequence of addresses:
Once: 63 through 70.
Loop ten times: 15 through 32; 80 through 95.

a. Suppose the cache is organized as direct mapped. Memory blocks 0, 4, and so on are assigned to line 1; blocks 1, 5, and so on to line 2; and so on. Compute the hit ratio.

b. Suppose the cache is organized as two-way set associative, with two sets of two lines each. Even-numbered blocks are assigned to set 0 and odd-numbered blocks are assigned to set 1. Compute the hit ratio for the two-way set-associative cache using the least recently used replacement scheme.

4.19 Consider a memory system with the following parameters:

$$T_c = 100 \text{ ns} \qquad C_c = 10^{-4} \text{ \$/bit}$$

$$T_m = 1200 \text{ ns} \qquad C_m = 10^{-5} \text{ \$/bit}$$

a. What is the cost of 1 MB of main memory?

b. What is the cost of 1 MB of main memory using cache memory technology?

c. If the effective access time is 10% greater than the cache access time, what is the hit ratio H?

4.20 **a.** Consider an L1 cache with an access time of 1 ns and a hit ratio of $H = 0.95$. Suppose that we can change the cache design (size of cache, cache organization) such that we increase H to 0.97, but increase access time to 1.5 ns. What conditions must be met for this change to result in improved performance?

d. Explain why this result makes intuitive sense.

4.21 Consider a single-level cache with an access time of 2.5 ns, a line size of 64 bytes, and a hit ratio of $H = 0.95$. Main memory uses a block transfer capability that has a first-word (4 bytes) access time of 50 ns and an access time of 5 ns for each word thereafter.

a. What is the access time when there is a cache miss? Assume that the cache waits until the line has been fetched from main memory and then re-executes for a hit.

b. Suppose that increasing the line size to 128 bytes increases the H to 0.97. Does this reduce the average memory access time?

4.22 A computer has a cache, main memory, and a disk used for virtual memory. If a referenced word is in the cache, 20 ns are required to access it. If it is in main memory but not in the cache, 60 ns are needed to load it into the cache, and then the reference is started again. If the word is not in main memory, 12 ms are required to fetch the word from disk, followed by 60 ns to copy it to the cache, and then the reference is started again. The cache hit ratio is 0.9 and the main memory hit ratio is 0.6. What is the average time in nanoseconds required to access a referenced word on this system?

4.23 Consider a cache with a line size of 64 bytes. Assume that on average 30% of the lines in the cache are dirty. A word consists of 8 bytes.

a. Assume there is a 3% miss rate (0.97 hit ratio). Compute the amount of main memory traffic, in terms of bytes per instruction for both write-through and write-back policies. Memory is read into cache one line at a time. However, for write back, a single word can be written from cache to main memory.

b. Repeat part a for a 5% rate.

c. Repeat part a for a 7% rate.

d. What conclusion can you draw from these results?

4.24 On the Motorola 68020 microprocessor, a cache access takes two clock cycles. Data access from main memory over the bus to the processor takes three clock cycles in the

case of no wait state insertion; the data are delivered to the processor in parallel with delivery to the cache.

a. Calculate the effective length of a memory cycle given a hit ratio of 0.9 and a clocking rate of 16.67 MHz.

b. Repeat the calculations assuming insertion of two wait states of one cycle each per memory cycle. What conclusion can you draw from the results?

4.25 Assume a processor having a memory cycle time of 300 ns and an instruction processing rate of 1 MIPS. On average, each instruction requires one bus memory cycle for instruction fetch and one for the operand it involves.

a. Calculate the utilization of the bus by the processor.

b. Suppose the processor is equipped with an instruction cache and the associated hit ratio is 0.5. Determine the impact on bus utilization.

4.26 The performance of a single-level cache system for a read operation can be characterized by the following equation:

$$T_a = T_c + (1 - H)T_m$$

where T_a is the average access time, T_c is the cache access time, T_m is the memory access time (memory to processor register), and H is the hit ratio. For simplicity, we assume that the word in question is loaded into the cache in parallel with the load to processor register. This is the same form as Equation (4.2).

a. Define T_b = time to transfer a line between cache and main memory, and W = fraction of write references. Revise the preceding equation to account for writes as well as reads, using a write-through policy.

b. Define W_b as the probability that a line in the cache has been altered. Provide an equation for T_a for the write-back policy.

4.27 For a system with two levels of cache, define T_{c_1} = first − level cache access time; T_{c_2} = second − level cache access time; T_m = memory access time; H_1 = first − level cache hit ratio; H_2 = combined first/second level cache hit ratio. Provide an equation for T_a for a read operation.

4.28 Assume the following performance characteristics on a cache read miss: one clock cycle to send an address to main memory and four clock cycles to access a 32-bit word from main memory and transfer it to the processor and cache.

a. If the cache line size is one word, what is the miss penalty (i.e., additional time required for a read in the event of a read miss)?

b. What is the miss penalty if the cache line size is four words and a multiple, nonburst transfer is executed?

c. What is the miss penalty if the cache line size is four words and a transfer is executed, with one clock cycle per word transfer?

4.29 For the cache design of the preceding problem, suppose that increasing the line size from one word to four words results in a decrease of the read miss rate from 3.2% to 1.1%. For both the nonburst transfer and the burst transfer case, what is the average miss penalty, averaged over all reads, for the two different line sizes?

APPENDIX 4A PERFORMANCE CHARACTERISTICS OF TWO-LEVEL MEMORIES

In this chapter, reference is made to a cache that acts as a buffer between main memory and processor, creating a two-level internal memory. This two-level architecture exploits a property known as locality to provide improved performance over a comparable one-level memory.

The main memory cache mechanism is part of the computer architecture, implemented in hardware and typically invisible to the operating system. There are two other instances of a two-level memory approach that also exploit locality and that are, at least partially, implemented in the operating system: virtual memory and the disk cache (Table 4.6). Virtual memory is explored in Chapter 8; disk cache is beyond the scope of this book but is examined in [STAL15]. In this appendix, we look at some of the performance characteristics of two-level memories that are common to all three approaches.

Locality

The basis for the performance advantage of a two-level memory is a principle known as **locality of reference** [DENN68]. This principle states that memory references tend to cluster. Over a long period of time, the clusters in use change, but over a short period of time, the processor is primarily working with fixed clusters of memory references.

Intuitively, the principle of locality makes sense. Consider the following line of reasoning:

1. Except for branch and call instructions, which constitute only a small fraction of all program instructions, program execution is sequential. Hence, in most cases, the next instruction to be fetched immediately follows the last instruction fetched.
2. It is rare to have a long uninterrupted sequence of procedure calls followed by the corresponding sequence of returns. Rather, a program remains confined to a rather narrow window of procedure-invocation depth. Thus, over a short period of time references to instructions tend to be localized to a few procedures.
3. Most iterative constructs consist of a relatively small number of instructions repeated many times. For the duration of the iteration, computation is therefore confined to a small contiguous portion of a program.
4. In many programs, much of the computation involves processing data structures, such as arrays or sequences of records. In many cases, successive references to these data structures will be to closely located data items.

Table 4.6 Characteristics of Two-Level Memories

	Main Memory Cache	Virtual Memory (paging)	Disk Cache
Typical access time ratios	5:1 (main memory vs. cache)	10^6:1 (main memory vs. disk)	10^6:1 (main memory vs. disk)
Memory management system	Implemented by special hardware	Combination of hardware and system software	System software
Typical block or page size	4 to 128 bytes (cache block)	64 to 4096 bytes (virtual memory page)	64 to 4096 bytes (disk block or pages)
Access of processor to second level	Direct access	Indirect access	Indirect access

Table 4.7 Relative Dynamic Frequency of High-Level Language Operations

Study	[HUCK83]	[KNUT71]	[PATT82a]		[TANE78]
Language	Pascal	FORTRAN	Pascal	C	SAL
Workload	Scientific	Student	System	System	System
Assign	74	67	45	38	42
Loop	4	3	5	3	4
Call	1	3	15	12	12
IF	20	11	29	43	36
GOTO	2	9	—	3	—
Other	—	7	6	1	6

This line of reasoning has been confirmed in many studies. With reference to point 1, a variety of studies have analyzed the behavior of high-level language programs. Table 4.7 includes key results, measuring the appearance of various statement types during execution, from the following studies. The earliest study of programming language behavior, performed by Knuth [KNUT71], examined a collection of FORTRAN programs used as student exercises. Tanenbaum [TANE78] published measurements collected from over 300 procedures used in operating-system programs and written in a language that supports structured programming (SAL). Patterson and Sequein [PATT82a] analyzed a set of measurements taken from compilers and programs for typesetting, computer-aided design (CAD), sorting, and file comparison. The programming languages C and Pascal were studied. Huck [HUCK83] analyzed four programs intended to represent a mix of general-purpose scientific computing, including fast Fourier transform and the integration of systems of differential equations. There is good agreement in the results of this mixture of languages and applications that branching and call instructions represent only a fraction of statements executed during the lifetime of a program. Thus, these studies confirm assertion 1.

With respect to assertion 2, studies reported in [PATT85a] provide confirmation. This is illustrated in Figure 4.20, which shows call-return behavior. Each call is represented by the line moving down and to the right, and each return by the line moving up and to the right. In the figure, a *window* with depth equal to 5 is defined. Only a sequence of calls and returns with a net movement of 6 in either direction causes the window to move. As can be seen, the executing program can remain within a stationary window for long periods of time. A study by the same analysts of C and Pascal programs showed that a window of depth 8 will need to shift only on less than 1% of the calls or returns [TAMI83].

A distinction is made in the literature between spatial locality and temporal locality. **Spatial locality** refers to the tendency of execution to involve a number of memory locations that are clustered. This reflects the tendency of a processor to access instructions sequentially. Spatial location also reflects the tendency of a program to access data locations sequentially, such as when processing a table of data. **Temporal locality** refers to the tendency for a processor to access memory locations that have been used recently. For example, when an iteration loop is executed, the processor executes the same set of instructions repeatedly.

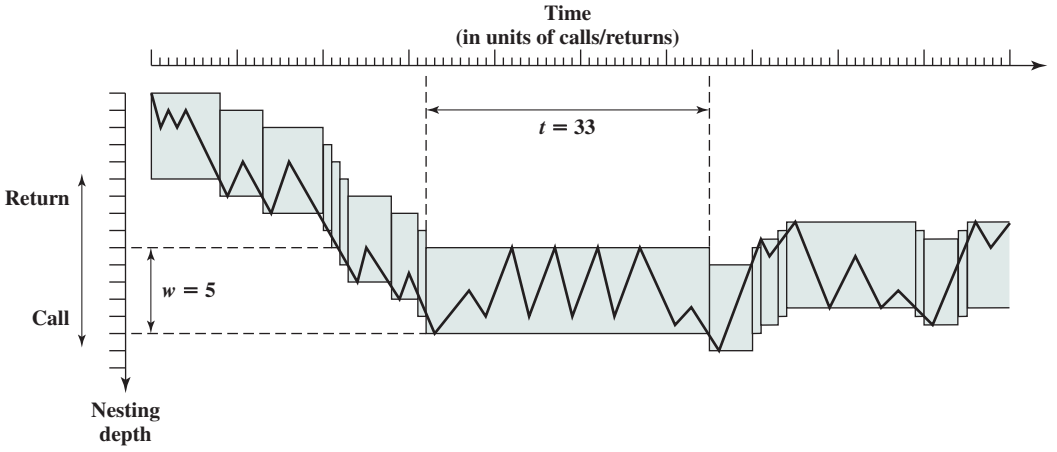


Figure 4.20 Example Call-Return Behavior of a Program

Traditionally, temporal locality is exploited by keeping recently used instruction and data values in cache memory and by exploiting a cache hierarchy. Spatial locality is generally exploited by using larger cache blocks and by incorporating prefetching mechanisms (fetching items of anticipated use) into the cache control logic. Recently, there has been considerable research on refining these techniques to achieve greater performance, but the basic strategies remain the same.

Operation of Two-Level Memory

The locality property can be exploited in the formation of a two-level memory. The upper-level memory (M1) is smaller, faster, and more expensive (per bit) than the lower-level memory (M2). M1 is used as a temporary store for part of the contents of the larger M2. When a memory reference is made, an attempt is made to access the item in M1. If this succeeds, then a quick access is made. If not, then a block of memory locations is copied from M2 to M1 and the access then takes place via M1. Because of locality, once a block is brought into M1, there should be a number of accesses to locations in that block, resulting in fast overall service.

To express the average time to access an item, we must consider not only the speeds of the two levels of memory, but also the probability that a given reference can be found in M1. We have

$$\begin{aligned} T_s &= H \times T_1 + (1 - H) \times (T_1 + T_2) \\ &= T_1 + (1 - H) \times T_2 \end{aligned} \tag{4.2}$$

where

T_s = average (system) access time
T_1 = access time of M1 (e.*g*., *cache*, *disk cache*)
T_2 = access time of M2 (e.*g*., *main memory*, *disk*)
H = hit ratio (fraction of time reference is found in M1)

Figure 4.2 shows average access time as a function of hit ratio. As can be seen, for a high percentage of hits, the average total access time is much closer to that of M1 than M2.

Performance

Let us look at some of the parameters relevant to an assessment of a two-level memory mechanism. First consider cost. We have

$$C_s = \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2} \tag{4.3}$$

where

C_s = average cost per bit for the combined two-level memory
C_1 = average cost per bit of upper-level memory M1
C_2 = average cost per bit of lower-level memory M2
S_1 = size of M1
S_2 = size of M2

We would like $C_s \approx C_2$. Given that $C_1 \gg C_2$, this requires $S_1 < S_2$. Figure 4.21 shows the relationship.

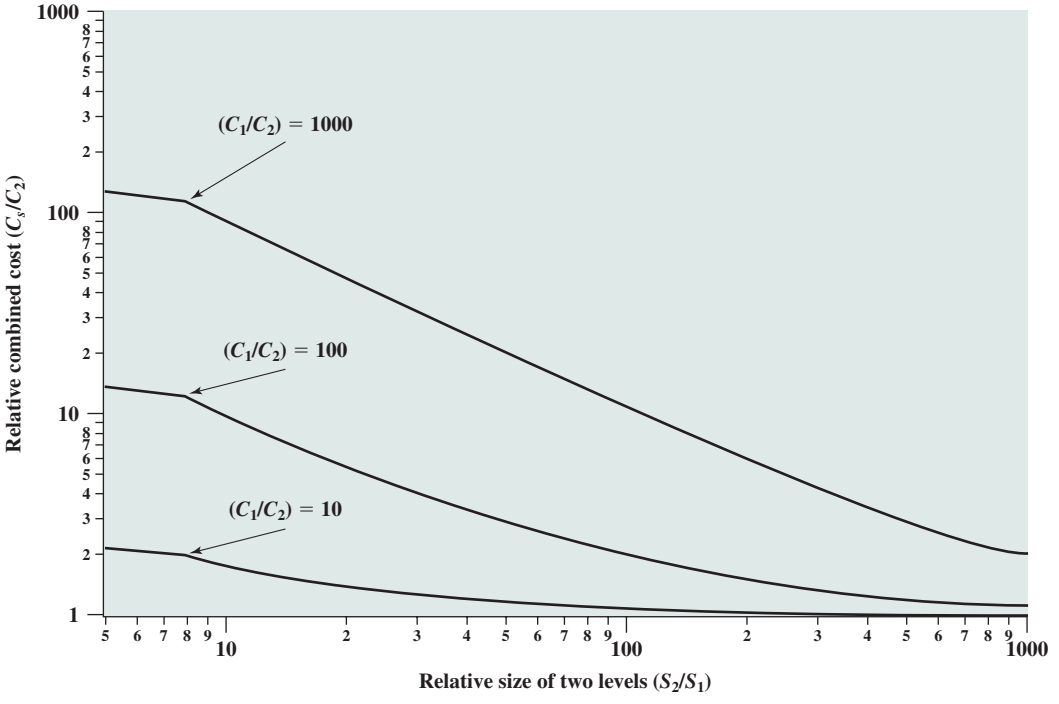


Figure 4.21 Relationship of Average Memory Cost to Relative Memory Size for a Two-Level Memory

Next, consider access time. For a two-level memory to provide a significant performance improvement, we need to have T_s approximately equal to T_1 ($T_s \approx T_1$). Given that T_1 is much less than T_2 ($T_1 << T_2$), a hit ratio of close to 1 is needed.

So we would like M1 to be small to hold down cost, and large to improve the hit ratio and therefore the performance. Is there a size of M1 that satisfies both requirements to a reasonable extent? We can answer this question with a series of subquestions:

- What value of hit ratio is needed so that $T_s \approx T_1$?
- What size of M1 will assure the needed hit ratio?
- Does this size satisfy the cost requirement?

To get at this, consider the quantity T_1/T_s, which is referred to as the *access efficiency*. It is a measure of how close average access time (T_s) is to M1 access time (T_1). From Equation (4.2),

$$\frac{T_1}{T_s} = \frac{1}{1 + (1 - H)\dfrac{T_2}{T_1}} \tag{4.4}$$

Figure 4.22 plots T_1/T_s as a function of the hit ratio H, with the quantity T_2/T_1 as a parameter. Typically, on-chip cache access time is about 25 to 50 times faster than main memory access time (i.e., T_2/T_1 is 25 to 50), off-chip cache access time

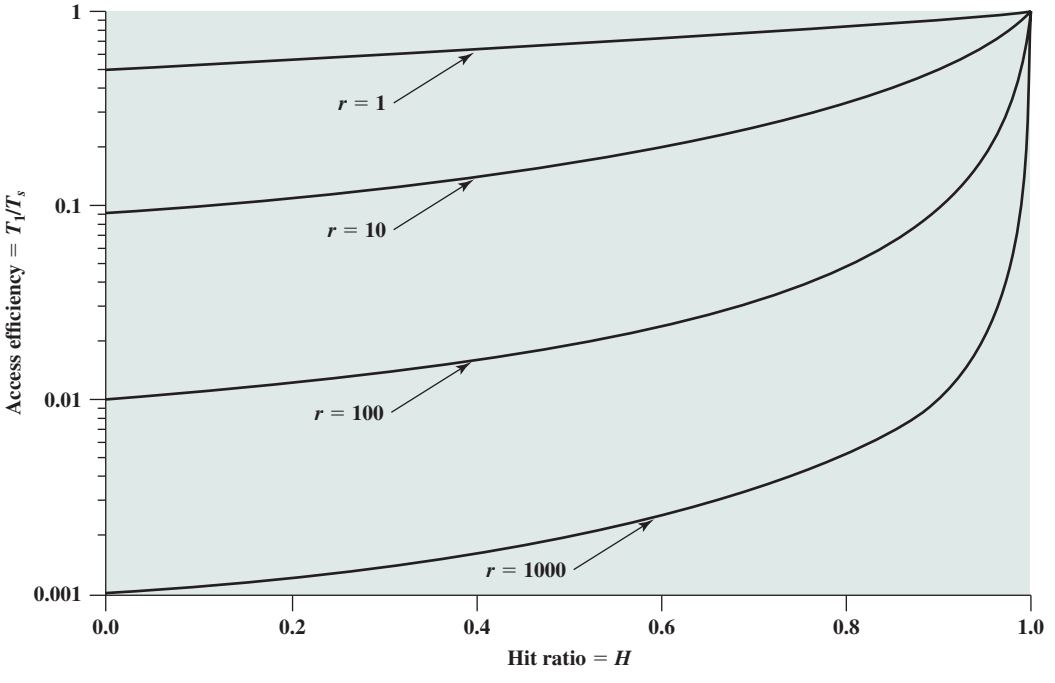


Figure 4.22 Access Efficiency as a Function of Hit Ratio ($r = T_2/T_1$)

is about 5 to 15 times faster than main memory access time (i.e., T_2/T_1 is 5 to 15), and main memory access time is about 1000 times faster than disk access time ($T_2/T_1 = 1000$). Thus, a hit ratio in the range of near 0.9 would seem to be needed to satisfy the performance requirement.

We can now phrase the question about relative memory size more exactly. Is a hit ratio of, say, 0.8 or better reasonable for $S_1 << S_2$? This will depend on a number of factors, including the nature of the software being executed and the details of the design of the two-level memory. The main determinant is, of course, the degree of locality. Figure 4.23 suggests the effect that locality has on the hit ratio. Clearly, if M1 is the same size as M2, then the hit ratio will be 1.0: All of the items in M2 are always also stored in M1. Now suppose that there is no locality; that is, references are completely random. In that case the hit ratio should be a strictly linear function of the relative memory size. For example, if M1 is half the size of M2, then at any time half of the items from M2 are also in M1 and the hit ratio will be 0.5. In practice, however, there is some degree of locality in the references. The effects of moderate and strong locality are indicated in the figure. Note that Figure 4.23 is not derived from any specific data or model; the figure suggests the type of performance that is seen with various degrees of locality.

So if there is strong locality, it is possible to achieve high values of hit ratio even with relatively small upper-level memory size. For example, numerous studies have shown that rather small cache sizes will yield a hit ratio above 0.75 *regardless of the size of main memory* (e.g., [AGAR89], [PRZY88], [STRE83], and [SMIT82]). A cache in the range of 1K to 128K words is generally adequate, whereas main

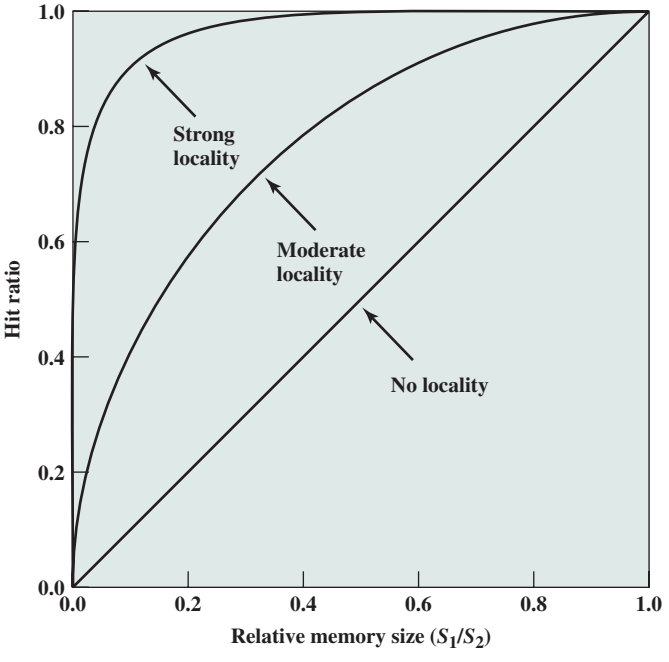


Figure 4.23 Hit Ratio as a Function of Relative Memory Size

memory is now typically in the gigabyte range. When we consider virtual memory and disk cache, we will cite other studies that confirm the same phenomenon, namely that a relatively small M1 yields a high value of hit ratio because of locality.

This brings us to the last question listed earlier: Does the relative size of the two memories satisfy the cost requirement? The answer is clearly yes. If we need only a relatively small upper-level memory to achieve good performance, then the average cost per bit of the two levels of memory will approach that of the cheaper lower-level memory.

Please note that with L2 cache, or even L2 and L3 caches, involved, analysis is much more complex. See [PEIR99] and [HAND98] for discussions.

CHAPTER 5

INTERNAL MEMORY

5.1 Semiconductor Main Memory
- Organization
- DRAM and SRAM
- Types of ROM
- Chip Logic
- Chip Packaging
- Module Organization
- Interleaved Memory

5.2 Error Correction

5.3 DDR DRAM
- Synchronous DRAM
- DDR SDRAM

5.4 Flash Memory
- Operation
- NOR and NAND Flash Memory

5.5 Newer Nonvolatile Solid-State Memory Technologies
- STT-RAM
- PCRAM
- ReRAM

5.6 Key Terms, Review Questions, and Problems

Learning Objectives

After studying this chapter, you should be able to:

- Present an overview of the principle types of semiconductor main memory.
- Understand the operation of a basic code that can detect and correct single-bit errors in 8-bit words.
- Summarize the properties of contemporary **DDR DRAM** organizations.
- Understand the difference between **NOR** and **NAND** flash memory.
- Present an overview of the newer nonvolatile solid-state memory technologies.

We begin this chapter with a survey of semiconductor main memory subsystems, including ROM, DRAM, and SRAM memories. Then we look at error control techniques used to enhance memory reliability. Following this, we look at more advanced DRAM architectures.

5.1 SEMICONDUCTOR MAIN MEMORY

In earlier computers, the most common form of random-access storage for computer main memory employed an array of doughnut-shaped ferromagnetic loops referred to as *cores*. Hence, main memory was often referred to as *core*, a term that persists to this day. The advent of, and advantages of, microelectronics has long since vanquished the magnetic core memory. Today, the use of semiconductor chips for main memory is almost universal. Key aspects of this technology are explored in this section.

Organization

The basic element of a **semiconductor memory** is the memory cell. Although a variety of electronic technologies are used, all semiconductor memory cells share certain properties:

- They exhibit two stable (or semistable) states, which can be used to represent binary 1 and 0.
- They are capable of being written into (at least once), to set the state.
- They are capable of being read to sense the state.

Figure 5.1 depicts the operation of a memory cell. Most commonly, the cell has three functional terminals capable of carrying an electrical signal. The select terminal, as the name suggests, selects a memory cell for a read or write operation. The control terminal indicates read or write. For writing, the other terminal provides an electrical signal that sets the state of the cell to 1 or 0. For reading, that terminal is used for output of the cell's state. The details of the internal organization, functioning, and timing of the memory cell depend on the specific integrated circuit technology used and are beyond the scope of this book, except for a brief summary. For our purposes, we will take it as given that individual cells can be selected for reading and writing operations.

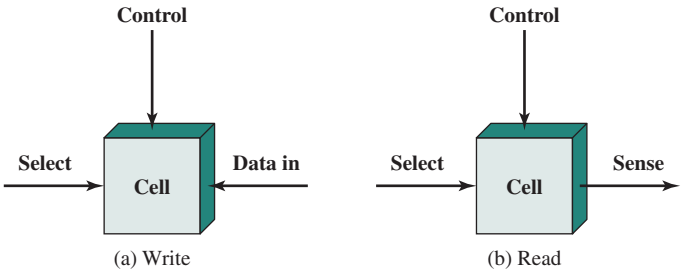


Figure 5.1 Memory Cell Operation

DRAM and SRAM

All of the memory types that we will explore in this chapter are random access. That is, individual words of memory are directly accessed through wired-in addressing logic.

Table 5.1 lists the major types of semiconductor memory. The most common is referred to as **random-access memory (RAM)**. This is, in fact, a misuse of the term, because all of the types listed in the table are random access. One distinguishing characteristic of memory that is designated as RAM is that it is possible both to read data from the memory and to write new data into the memory easily and rapidly. Both the reading and writing are accomplished through the use of electrical signals.

The other distinguishing characteristic of traditional RAM is that it is volatile. A RAM must be provided with a constant power supply. If the power is interrupted, then the data are lost. Thus, RAM can be used only as temporary storage. The two traditional forms of RAM used in computers are DRAM and SRAM. Newer forms of RAM, discussed in Section 5.5, are nonvolatile.

DYNAMIC RAM RAM technology is divided into two technologies: dynamic and static. A **dynamic RAM (DRAM)** is made with cells that store data as charge on capacitors. The presence or absence of charge in a capacitor is interpreted as a binary 1 or 0. Because capacitors have a natural tendency to discharge, dynamic RAMs require periodic charge refreshing to maintain data storage. The term

Table 5.1 Semiconductor Memory Types

Memory Type	Category	Erasure	Write Mechanism	Volatility
Random-access memory (RAM)	Read-write memory	Electrically, byte-level	Electrically	Volatile
Read-only memory (ROM)	Read-only memory	Not possible	Masks	Nonvolatile
Programmable ROM (PROM)			Electrically	
Erasable PROM (EPROM)	Read-mostly memory	UV light, chip-level		
Electrically Erasable PROM (EEPROM)		Electrically, byte-level		
Flash memory		Electrically, block-level		

dynamic refers to this tendency of the stored charge to leak away, even with power continuously applied.

Figure 5.2a is a typical DRAM structure for an individual cell that stores one bit. The address line is activated when the bit value from this cell is to be read or written. The transistor acts as a switch that is closed (allowing current to flow) if a voltage is applied to the address line and open (no current flows) if no voltage is present on the address line.

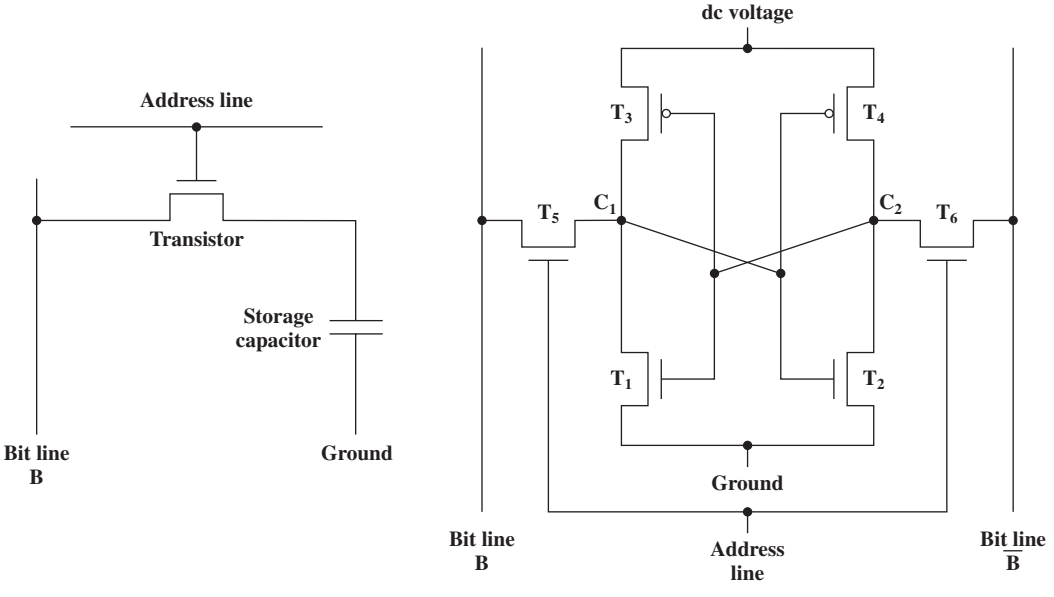
For the write operation, a voltage signal is applied to the bit line; a high voltage represents 1, and a low voltage represents 0. A signal is then applied to the address line, allowing a charge to be transferred to the capacitor.

For the read operation, when the address line is selected, the transistor turns on and the charge stored on the capacitor is fed out onto a bit line and to a sense amplifier. The sense amplifier compares the capacitor voltage to a reference value and determines if the cell contains a logic 1 or a logic 0. The readout from the cell discharges the capacitor, which must be restored to complete the operation.

Although the DRAM cell is used to store a single bit (0 or 1), it is essentially an analog device. The capacitor can store any charge value within a range; a threshold value determines whether the charge is interpreted as 1 or 0.

STATIC RAM In contrast, a **static RAM (SRAM)** is a digital device that uses the same logic elements used in the processor. In a SRAM, binary values are stored using traditional flip-flop logic-gate configurations (see Chapter 11 for a description of flip-flops). A static RAM will hold its data as long as power is supplied to it.

Figure 5.2b is a typical SRAM structure for an individual cell. Four transistors (T_1, T_2, T_3, T_4) are cross connected in an arrangement that produces a stable logic



(a) Dynamic RAM (DRAM) cell

(b) Static RAM (SRAM) cell

Figure 5.2 Typical Memory Cell Structures

state. In logic state 1, point C_1 is high and point C_2 is low; in this state, T_1 and T_4 are off and T_2 and T_3 are on.[1] In logic state 0, point C_1 is low and point C_2 is high; in this state, T_1 and T_4 are on and T_2 and T_3 are off. Both states are stable as long as the direct current (dc) voltage is applied. Unlike the DRAM, no refresh is needed to retain data.

As in the DRAM, the SRAM address line is used to open or close a switch. The address line controls two transistors (T_5 and T_6). When a signal is applied to this line, the two transistors are switched on, allowing a read or write operation. For a write operation, the desired bit value is applied to line B, while its complement is applied to line $\overline{B}$. This forces the four transistors (T_1, T_2, T_3, T_4) into the proper state. For a read operation, the bit value is read from line B.

SRAM VERSUS DRAM Both static and dynamic RAMs are volatile; that is, power must be continuously supplied to the memory to preserve the bit values. A dynamic memory cell is simpler and smaller than a static memory cell. Thus, a DRAM is more dense (smaller cells = more cells per unit area) and less expensive than a corresponding SRAM. On the other hand, a DRAM requires the supporting refresh circuitry. For larger memories, the fixed cost of the refresh circuitry is more than compensated for by the smaller variable cost of DRAM cells. Thus, DRAMs tend to be favored for large memory requirements. A final point is that SRAMs are somewhat faster than DRAMs. Because of these relative characteristics, SRAM is used for cache memory (both on and off chip), and DRAM is used for main memory.

Types of ROM

As the name suggests, a **read-only memory (ROM)** contains a permanent pattern of data that cannot be changed. A ROM is nonvolatile; that is, no power source is required to maintain the bit values in memory. While it is possible to read a ROM, it is not possible to write new data into it. An important application of ROMs is microprogramming, discussed in Part Four. Other potential applications include

- Library subroutines for frequently wanted functions
- System programs
- Function tables

For a modest-sized requirement, the advantage of ROM is that the data or program is permanently in main memory and need never be loaded from a secondary storage device.

A ROM is created like any other integrated circuit chip, with the data actually wired into the chip as part of the fabrication process. This presents two problems:

- The data insertion step includes a relatively large fixed cost, whether one or thousands of copies of a particular ROM are fabricated.
- There is no room for error. If one bit is wrong, the whole batch of ROMs must be thrown out.

When only a small number of ROMs with a particular memory content is needed, a less expensive alternative is the **programmable ROM (PROM)**. Like the

[1]The circles associated with T_3 and T_4 in Figure 5.2b indicate signal negation.

ROM, the PROM is **nonvolatile** and may be written into only once. For the PROM, the writing process is performed electrically and may be performed by a supplier or customer at a time later than the original chip fabrication. Special equipment is required for the writing or "programming" process. PROMs provide flexibility and convenience. The ROM remains attractive for high-volume production runs.

Another variation on read-only memory is the **read-mostly memory**, which is useful for applications in which read operations are far more frequent than write operations but for which nonvolatile storage is required. There are three common forms of read-mostly memory: EPROM, EEPROM, and flash memory.

The optically **erasable programmable read-only memory (EPROM)** is read and written electrically, as with PROM. However, before a write operation, all the storage cells must be erased to the same initial state by exposure of the packaged chip to ultraviolet radiation. Erasure is performed by shining an intense ultraviolet light through a window that is designed into the memory chip. This erasure process can be performed repeatedly; each erasure can take as much as 20 minutes to perform. Thus, the EPROM can be altered multiple times and, like the ROM and PROM, holds its data virtually indefinitely. For comparable amounts of storage, the EPROM is more expensive than PROM, but it has the advantage of the multiple update capability.

A more attractive form of read-mostly memory is **electrically erasable programmable read-only memory (EEPROM)**. This is a read-mostly memory that can be written into at any time without erasing prior contents; only the byte or bytes addressed are updated. The write operation takes considerably longer than the read operation, on the order of several hundred microseconds per byte. The EEPROM combines the advantage of nonvolatility with the flexibility of being updatable in place, using ordinary bus control, address, and data lines. EEPROM is more expensive than EPROM and also is less dense, supporting fewer bits per chip.

Another form of semiconductor memory is **flash memory** (so named because of the speed with which it can be reprogrammed). First introduced in the mid-1980s, flash memory is intermediate between EPROM and EEPROM in both cost and functionality. Like EEPROM, flash memory uses an electrical erasing technology. An entire flash memory can be erased in one or a few seconds, which is much faster than EPROM. In addition, it is possible to erase just blocks of memory rather than an entire chip. Flash memory gets its name because the microchip is organized so that a section of memory cells are erased in a single action or "flash." However, flash memory does not provide byte-level erasure. Like EPROM, flash memory uses only one transistor per bit, and so achieves the high density (compared with EEPROM) of EPROM.

Chip Logic

As with other integrated circuit products, semiconductor memory comes in packaged chips (Figure 1.11). Each chip contains an array of memory cells.

In the memory hierarchy as a whole, we saw that there are trade-offs among speed, density, and cost. These trade-offs also exist when we consider the organization of memory cells and functional logic on a chip. For semiconductor memories, one of the key design issues is the number of bits of data that may be read/written at a time. At one extreme is an organization in which the physical arrangement of cells in the array is the same as the logical arrangement (as perceived by the processor) of words in memory. The array is organized into W words of B bits each.

For example, a 16-Mbit chip could be organized as 1M 16-bit words. At the other extreme is the so-called 1-bit-per-chip organization, in which data are read/written one bit at a time. We will illustrate memory chip organization with a DRAM; ROM organization is similar, though simpler.

Figure 5.3 shows a typical organization of a 16-Mbit DRAM. In this case, 4 bits are read or written at a time. Logically, the memory array is organized as four square arrays of 2048 by 2048 elements. Various physical arrangements are possible. In any case, the elements of the array are connected by both horizontal (row) and vertical (column) lines. Each horizontal line connects to the Select terminal of each cell in its row; each vertical line connects to the Data-In/Sense terminal of each cell in its column.

Address lines supply the address of the word to be selected. A total of $\log_2 W$ lines are needed. In our example, 11 address lines are needed to select one of 2048 rows. These 11 lines are fed into a row decoder, which has 11 lines of input and 2048 lines for output. The logic of the decoder activates a single one of the 2048 outputs depending on the bit pattern on the 11 input lines ($2^{11} = 2048$).

An additional 11 address lines select one of 2048 columns of 4 bits per column. Four data lines are used for the input and output of 4 bits to and from a data buffer. On input (write), the bit driver of each bit line is activated for a 1 or 0 according to the value of the corresponding data line. On output (read), the value of each bit line is passed through a sense amplifier and presented to the data lines. The row line selects which row of cells is used for reading or writing.

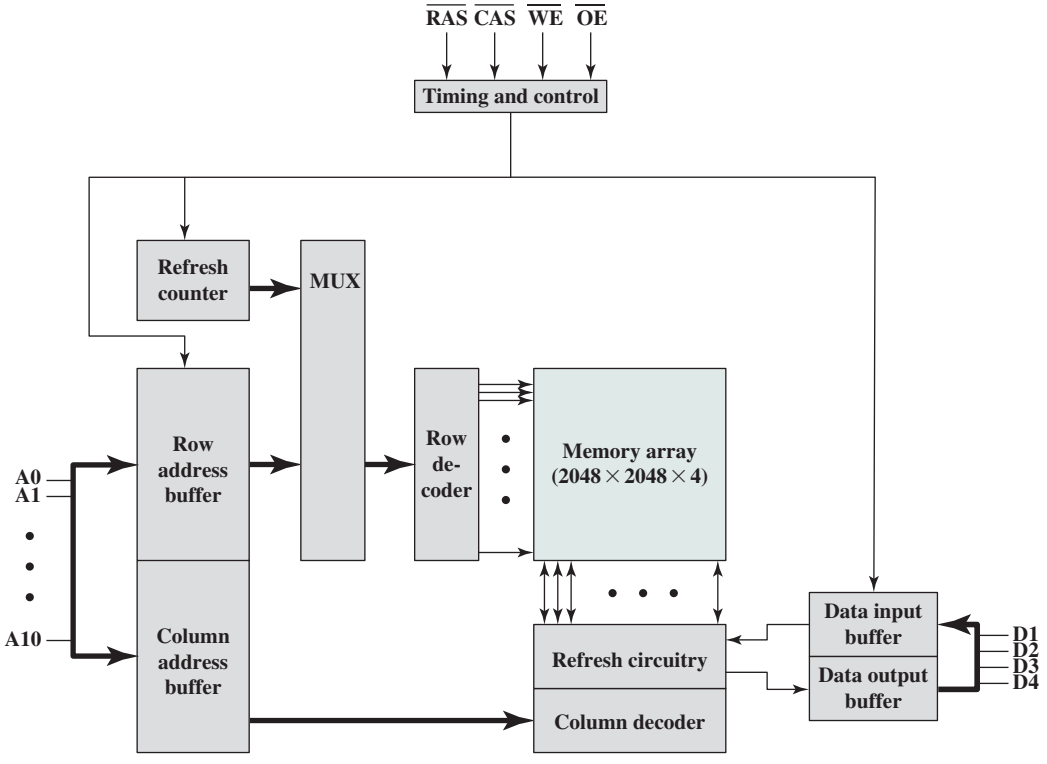


Figure 5.3 Typical 16-Mbit DRAM (4M × 4)

Because only 4 bits are read/written to this DRAM, there must be multiple DRAMs connected to the memory controller to read/write a word of data to the bus.

Note that there are only 11 address lines (A0–A10), half the number you would expect for a 2048 × 2048 array. This is done to save on the number of pins. The 22 required address lines are passed through select logic external to the chip and multiplexed onto the 11 address lines. First, 11 address signals are passed to the chip to define the row address of the array, and then the other 11 address signals are presented for the column address. These signals are accompanied by row address select ($\overline{\text{RAS}}$) and column address select ($\overline{\text{CAS}}$) signals to provide timing to the chip.

The write enable ($\overline{\text{WE}}$) and output enable ($\overline{\text{OE}}$) pins determine whether a write or read operation is performed. Two other pins, not shown in Figure 5.3, are ground (Vss) and a voltage source (Vcc).

As an aside, multiplexed addressing plus the use of square arrays result in a quadrupling of memory size with each new generation of memory chips. One more pin devoted to addressing doubles the number of rows and columns, and so the size of the chip memory grows by a factor of 4.

Figure 5.3 also indicates the inclusion of refresh circuitry. All DRAMs require a refresh operation. A simple technique for refreshing is, in effect, to disable the DRAM chip while all data cells are refreshed. The refresh counter steps through all of the row values. For each row, the output lines from the refresh counter are supplied to the row decoder and the RAS line is activated. The data are read out and written back into the same location. This causes each cell in the row to be refreshed.

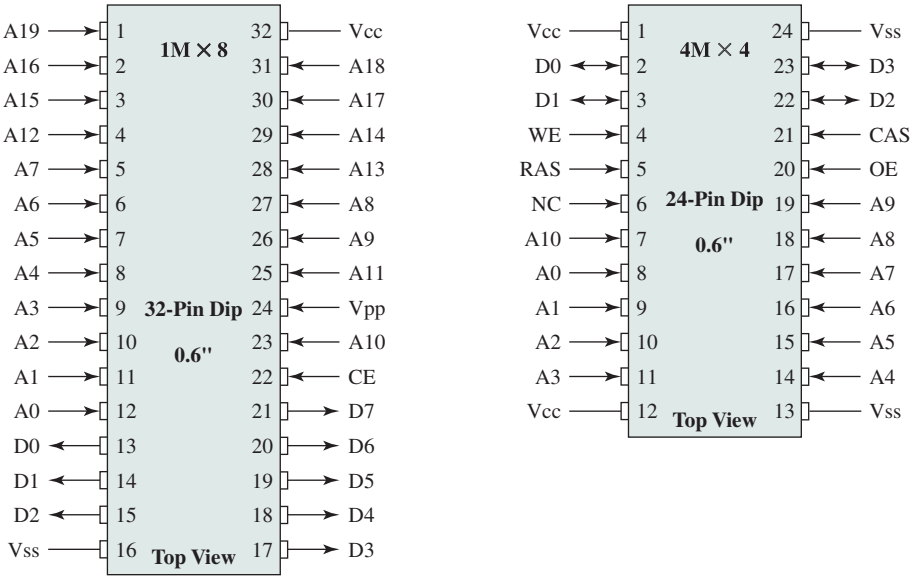
Chip Packaging

As was mentioned in Chapter 2, an integrated circuit is mounted on a package that contains pins for connection to the outside world.

Figure 5.4a shows an example EPROM package, which is an 8-Mbit chip organized as 1M × 8. In this case, the organization is treated as a one-word-per-chip package. The package includes 32 pins, which is one of the standard chip package sizes. The pins support the following signal lines:

- The address of the word being accessed. For 1M words, a total of 20 ($2^{20} = 1\text{M}$) pins are needed (A0–A19).
- The data to be read out, consisting of 8 lines (D0–D7).
- The power supply to the chip (V_{cc}).
- A ground pin (V_{ss}).
- A chip enable (CE) pin. Because there may be more than one memory chip, each of which is connected to the same address bus, the CE pin is used to indicate whether or not the address is valid for this chip. The CE pin is activated by logic connected to the higher-order bits of the address bus (i.e., address bits above A19). The use of this signal is illustrated presently.
- A program voltage (V_{pp}) that is supplied during programming (write operations).

A typical DRAM pin configuration is shown in Figure 5.4b, for a 16-Mbit chip organized as 4M × 4. There are several differences from a ROM chip. Because a RAM can be updated, the data pins are input/output. The write enable (WE) and output enable (OE) pins indicate whether this is a write or read operation.



(a) 8-Mbit EPROM

(b) 16-Mbit DRAM

Figure 5.4 Typical Memory Package Pins and Signals

Because the DRAM is accessed by row and column, and the address is multiplexed, only 11 address pins are needed to specify the 4M row/column combinations ($2^{11} \times 2^{11} = 2^{22} = 4M$). The functions of the row address select (RAS) and column address select (CAS) pins were discussed previously. Finally, the no connect (NC) pin is provided so that there are an even number of pins.

Module Organization

If a RAM chip contains only one bit per word, then clearly we will need at least a number of chips equal to the number of bits per word. As an example, Figure 5.5 shows how a memory module consisting of 256K 8-bit words could be organized. For 256K words, an 18-bit address is needed and is supplied to the module from some external source (e.g., the address lines of a bus to which the module is attached). The address is presented to 8 256K $\times$ 1-bit chips, each of which provides the input/output of one bit.

This organization works as long as the size of memory equals the number of bits per chip. In the case in which larger memory is required, an array of chips is needed. Figure 5.6 shows the possible organization of a memory consisting of 1M word by 8 bits per word. In this case, we have four columns of chips, each column containing 256K words arranged as in Figure 5.5. For 1M word, 20 address lines are needed. The 18 least significant bits are routed to all 32 modules. The high-order 2 bits are input to a group select logic module that sends a chip enable signal to one of the four columns of modules.

Interleaved Memory

Main memory is composed of a collection of DRAM memory chips. A number of chips can be grouped together to form a *memory bank*. It is possible to organize the memory

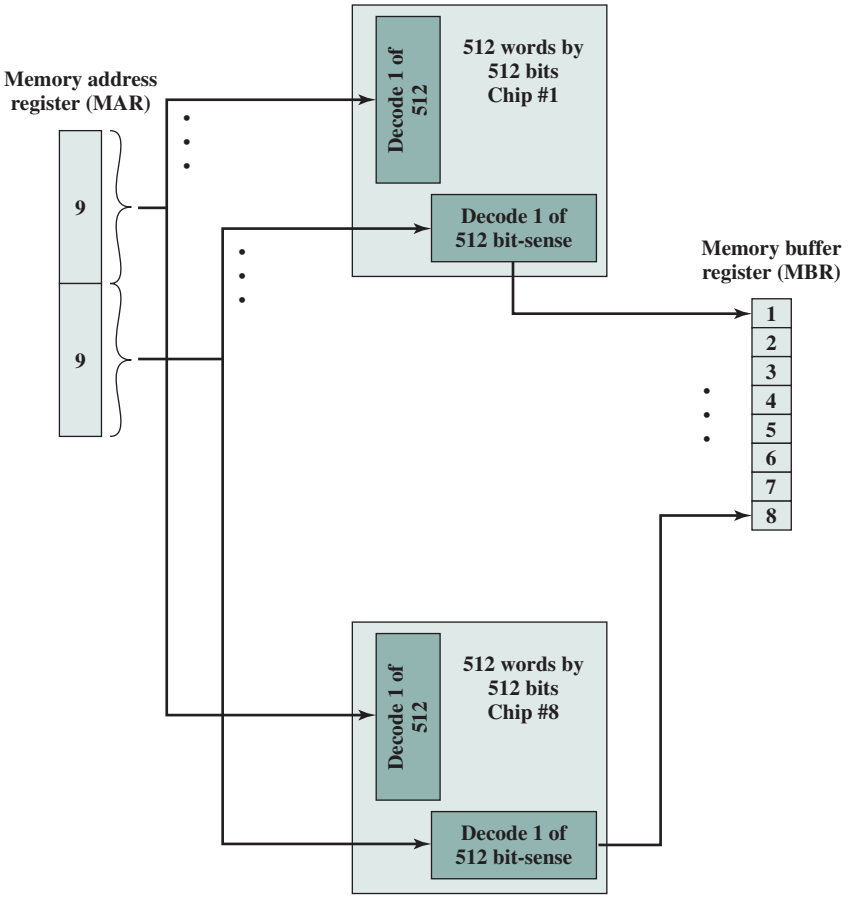


Figure 5.5 256-KByte Memory Organization

banks in a way known as interleaved memory. Each bank is independently able to service a memory read or write request, so that a system with K banks can service K requests simultaneously, increasing memory read or write rates by a factor of K. If consecutive words of memory are stored in different banks, then the transfer of a block of memory is speeded up. Appendix G explores the topic of interleaved memory.

Interleaved Memory Simulator

5.2 ERROR CORRECTION

A semiconductor memory system is subject to errors. These can be categorized as hard failures and soft errors. A **hard failure** is a permanent physical defect so that the memory cell or cells affected cannot reliably store data but become stuck at 0 or 1 or

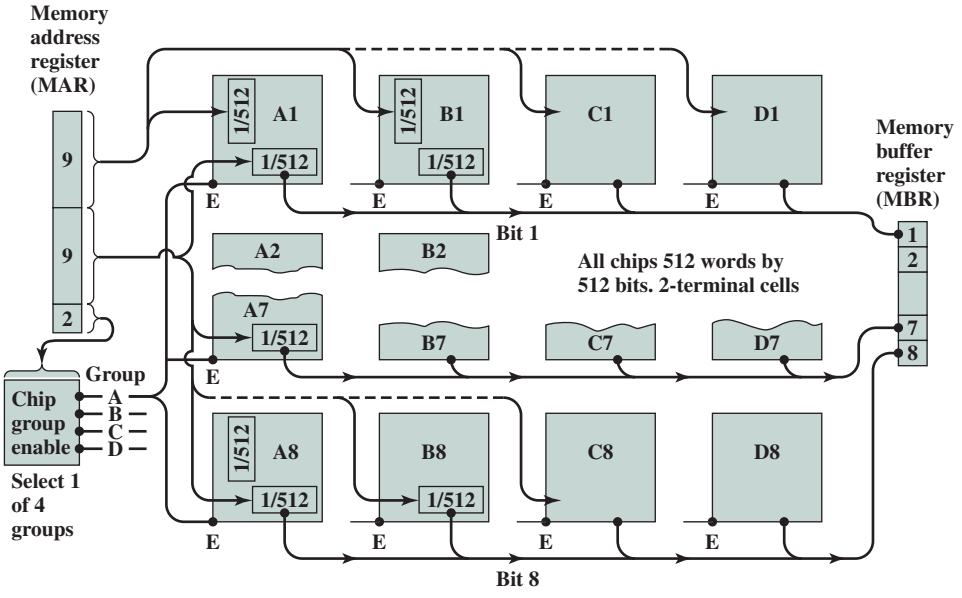


Figure 5.6 1-MB Memory Organization

switch erratically between 0 and 1. Hard errors can be caused by harsh environmental abuse, manufacturing defects, and wear. A **soft error** is a random, nondestructive event that alters the contents of one or more memory cells without damaging the memory. Soft errors can be caused by power supply problems or alpha particles. These particles result from radioactive decay and are distressingly common because radioactive nuclei are found in small quantities in nearly all materials. Both hard and soft errors are clearly undesirable, and most modern main memory systems include logic for both detecting and correcting errors.

Figure 5.7 illustrates in general terms how the process is carried out. When data are to be written into memory, a calculation, depicted as a function f, is performed on the data to produce a code. Both the code and the data are stored. Thus, if an M-bit word of data is to be stored and the code is of length K bits, then the actual size of the stored word is $M + K$ bits.

When the previously stored word is read out, the code is used to detect and possibly correct errors. A new set of K code bits is generated from the M data bits and compared with the fetched code bits. The comparison yields one of three results:

- No errors are detected. The fetched data bits are sent out.
- An error is detected, and it is possible to correct the error. The data bits plus **error correction** bits are fed into a corrector, which produces a corrected set of M bits to be sent out.
- An error is detected, but it is not possible to correct it. This condition is reported.

Codes that operate in this fashion are referred to as **error-correcting codes**. A code is characterized by the number of bit errors in a word that it can correct and detect.

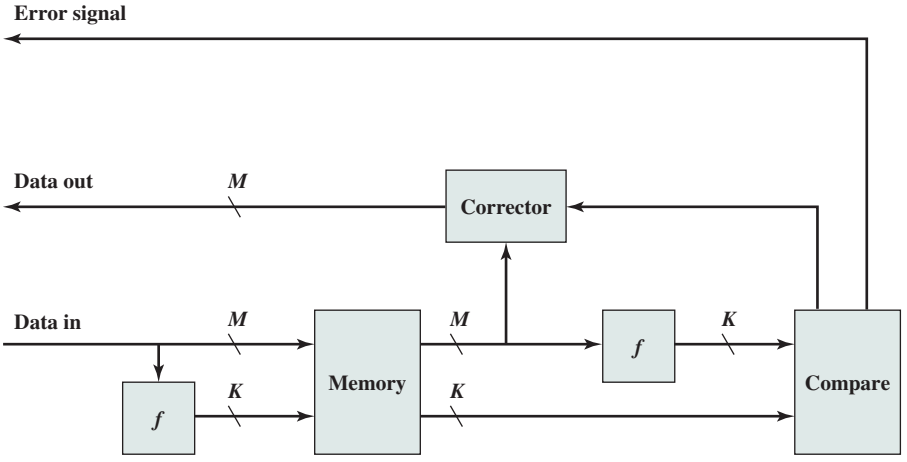


Figure 5.7 Error-Correcting Code Function

The simplest of the error-correcting codes is the **Hamming code** devised by Richard Hamming at Bell Laboratories. Figure 5.8 uses Venn diagrams to illustrate the use of this code on 4-bit words ($M = 4$). With three intersecting circles, there are seven compartments. We assign the 4 data bits to the inner compartments (Figure 5.8a). The remaining compartments are filled with what are called *parity bits.* Each parity bit is chosen so that the total number of 1s in its circle is even (Figure 5.8b). Thus, because circle A includes three data 1s, the parity bit in that circle is set to 1. Now, if an error changes one of the data bits (Figure 5.8c), it is easily found. By checking the parity bits, discrepancies are found in circle A and circle C but not in circle B. Only one of the seven compartments is in A and C but not B (Figure 5.8d). The error can therefore be corrected by changing that bit.

To clarify the concepts involved, we will develop a code that can detect and correct single-bit errors in 8-bit words.

To start, let us determine how long the code must be. Referring to Figure 5.7, the comparison logic receives as input two K-bit values. A bit-by-bit comparison is done by taking the exclusive-OR of the two inputs. The result is called the *syndrome word.* Thus, each bit of the **syndrome** is 0 or 1 according to if there is or is not a match in that bit position for the two inputs.

The syndrome word is therefore K bits wide and has a range between 0 and $2^K - 1$. The value 0 indicates that no error was detected, leaving $2^K - 1$ values to indicate, if there is an error, which bit was in error. Now, because an error could occur on any of the M data bits or K check bits, we must have

$$2^K - 1 \geq M + K$$

This inequality gives the number of bits needed to correct a single bit error in a word containing M data bits. For example, for a word of 8 data bits ($M = 8$), we have

- $K = 3: 2^3 - 1 < 8 + 3$
- $K = 4: 2^4 - 1 > 8 + 4$

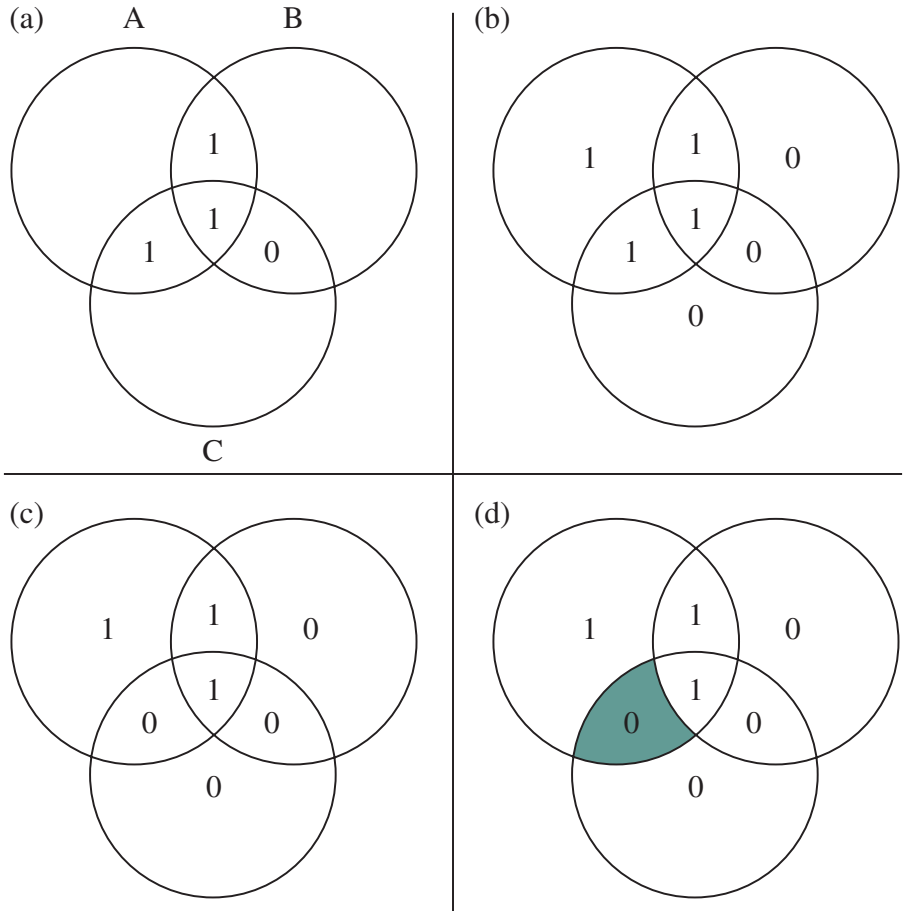


Figure 5.8 Hamming Error-Correcting Code

Thus, eight data bits require four check bits. The first three columns of Table 5.2 lists the number of check bits required for various data word lengths.

For convenience, we would like to generate a 4-bit syndrome for an 8-bit data word with the following characteristics:

- If the syndrome contains all 0s, no error has been detected.
- If the syndrome contains one and only one bit set to 1, then an error has occurred in one of the 4 check bits. No correction is needed.
- If the syndrome contains more than one bit set to 1, then the numerical value of the syndrome indicates the position of the data bit in error. This data bit is inverted for correction.

To achieve these characteristics, the data and check bits are arranged into a 12-bit word as depicted in Figure 5.9. The bit positions are numbered from 1 to 12. Those bit positions whose position numbers are powers of 2 are designated as check

Table 5.2 Increase in Word Length with Error Correction

	Single-Error Correction		Single-Error Correction/ Double-Error Detection	
Data Bits	**Check Bits**	**% Increase**	**Check Bits**	**% Increase**
8	4	50.0	5	62.5
16	5	31.25	6	37.5
32	6	18.75	7	21.875
64	7	10.94	8	12.5
128	8	6.25	9	7.03
256	9	3.52	10	3.91

bits. The check bits are calculated as follows, where the symbol $\oplus$ designates the exclusive-OR operation:

$$
\begin{aligned}
C1 &= D1 \oplus D2 \oplus D4 \oplus D5 \oplus D7 \\
C2 &= D1 \oplus D3 \oplus D4 \oplus D6 \oplus D7 \\
C4 &= D2 \oplus D3 \oplus D4 \oplus D8 \\
C8 &= \oplus D5 \oplus D6 \oplus D7 \oplus D8
\end{aligned}
$$

Each check bit operates on every data bit whose position number contains a 1 in the same bit position as the position number of that check bit. Thus, data bit positions 3, 5, 7, 9, and 11 (D1, D2, D4, D5, D7) all contain a 1 in the least significant bit of their position number as does C1; bit positions 3, 6, 7, 10, and 11 all contain a 1 in the second bit position, as does C2; and so on. Looked at another way, bit position n is checked by those bits C_i such that $\sum_i = n$. For example, position 7 is checked by bits in position 4, 2, and 1; and $7 = 4 + 2 + 1$.

Let us verify that this scheme works with an example. Assume that the 8-bit input word is 00111001, with data bit D1 in the rightmost position. The calculations are as follows:

$$
\begin{aligned}
C1 &= 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1 \\
C2 &= 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1 \\
C4 &= 0 \oplus 0 \oplus 1 \oplus 0 = 1 \\
C8 &= 1 \oplus 1 \oplus 0 \oplus 0 = 0
\end{aligned}
$$

Bit position	12	11	10	9	8	7	6	5	4	3	2	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Data bit	D8	D7	D6	D5		D4	D3	D2		D1		
Check bit					C8				C4		C2	C1

Figure 5.9 Layout of Data Bits and Check Bits

Suppose now that data bit 3 sustains an error and is changed from 0 to 1. When the check bits are recalculated, we have

$$C1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C2 = 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 0$$
$$C4 = 0 \oplus 1 \oplus 1 \oplus 0 = 0$$
$$C8 = 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

When the new check bits are compared with the old check bits, the syndrome word is formed:

	C8	C4	C2	C1
	0	1	1	1
$\oplus$	0	0	0	1
	0	1	1	0

The result is 0110, indicating that bit position 6, which contains data bit 3, is in error.

Figure 5.10 illustrates the preceding calculation. The data and check bits are positioned properly in the 12-bit word. Four of the data bits have a value 1 (shaded in the table), and their bit position values are XORed to produce the Hamming code 0111, which forms the four check digits. The entire block that is stored is 001101001111. Suppose now that data bit 3, in bit position 6, sustains an error and is changed from 0 to 1. The resulting block is 001101101111, with a Hamming code of 0001. An XOR of the Hamming code and all of the bit position values for nonzero data bits results in 0110. The nonzero result detects an error and indicates that the error is in bit position 6.

The code just described is known as a **single-error-correcting (SEC) code**. More commonly, semiconductor memory is equipped with a **single-error-correcting, double-error-detecting (SEC-DED) code**. As Table 5.2 shows, such codes require one additional bit compared with SEC codes.

Figure 5.11 illustrates how such a code works, again with a 4-bit data word. The sequence shows that if two errors occur (Figure 5.11c), the checking procedure goes astray (d) and worsens the problem by creating a third error (e). To overcome

Bit position	12	11	10	9	8	7	6	5	4	3	2	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Data bit	D8	D7	D6	D5		D4	D3	D2		D1		
Check bit					C8				C4		C2	C1
Word stored as	0	0	1	1	0	1	0	0	1	1	1	1
Word fetched as	0	0	1	1	0	1	1	0	1	1	1	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Check bit					0				0		0	1

Figure 5.10 Check Bit Calculation

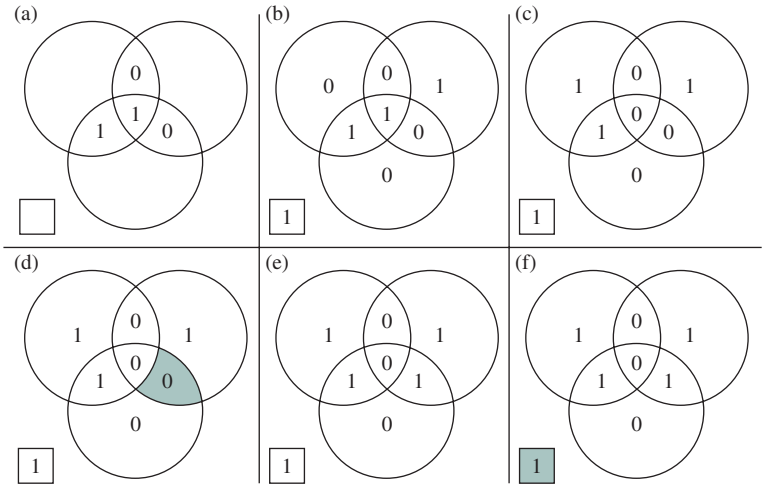


Figure 5.11 Hamming SEC-DEC Code

the problem, an eighth bit is added that is set so that the total number of 1s in the diagram is even. The extra parity bit catches the error (f).

An error-correcting code enhances the reliability of the memory at the cost of added complexity. With a 1-bit-per-chip organization, an SEC-DED code is generally considered adequate. For example, the IBM 30xx implementations used an 8-bit SEC-DED code for each 64 bits of data in main memory. Thus, the size of main memory is actually about 12% larger than is apparent to the user. The VAX computers used a 7-bit SEC-DED for each 32 bits of memory, for a 22% overhead. Contemporary DRAM systems may have anywhere from 7% to 20% overhead [SHAR03].

5.3 DDR DRAM

As discussed in Chapter 1, one of the most critical system bottlenecks when using high-performance processors is the interface to internal main memory. This interface is the most important pathway in the entire computer system. The basic building block of main memory remains the DRAM chip, as it has for decades; until recently, there had been no significant changes in DRAM architecture since the early 1970s. The traditional DRAM chip is constrained both by its internal architecture and by its interface to the processor's memory bus.

We have seen that one attack on the performance problem of DRAM main memory has been to insert one or more levels of high-speed SRAM cache between the DRAM main memory and the processor. But SRAM is much costlier than DRAM, and expanding cache size beyond a certain point yields diminishing returns.

In recent years, a number of enhancements to the basic DRAM architecture have been explored. The schemes that currently dominate the market are SDRAM and DDR-DRAM. We examine each of these in turn.

Synchronous DRAM

One of the most widely used forms of DRAM is the **synchronous DRAM (SDRAM)**. Unlike the traditional DRAM, which is asynchronous, the SDRAM exchanges data with the processor synchronized to an external clock signal and running at the full speed of the processor/memory bus without imposing wait states.

In a typical DRAM, the processor presents addresses and control levels to the memory, indicating that a set of data at a particular location in memory should be either read from or written into the DRAM. After a delay, the access time, the DRAM either writes or reads the data. During the access-time delay, the DRAM performs various internal functions, such as activating the high capacitance of the row and column lines, sensing the data, and routing the data out through the output buffers. The processor must simply wait through this delay, slowing system performance.

With synchronous access, the DRAM moves data in and out under control of the system clock. The processor or other master issues the instruction and address information, which is latched by the DRAM. The DRAM then responds after a set number of clock cycles. Meanwhile, the master can safely do other tasks while the SDRAM is processing the request.

Figure 5.12 shows the internal logic of a typical 256-Mb SDRAM typical of SDRAM organization, and Table 5.3 defines the various pin assignments. The

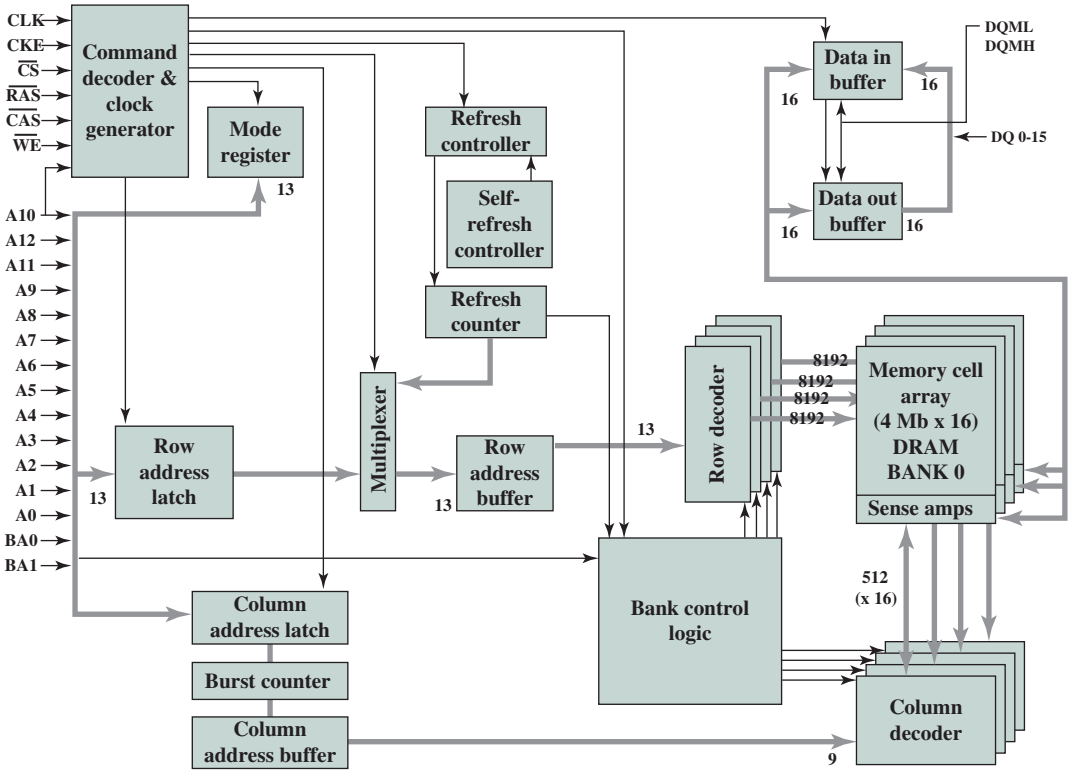


Figure 5.12 256-Mb Synchronous Dynamic RAM (SDRAM)

Table 5.3 SDRAM Pin Assignments

A0 to A13	Address inputs
BA0, BA1	Bank address lines
CLK	Clock input
CKE	Clock enable
$\overline{CS}$	Chip select
$\overline{RAS}$	Row address strobe
$\overline{CAS}$	Column address strobe
$\overline{WE}$	Write enable
DQ0 to DQ7	Data input/output
DQM	Data mask

SDRAM employs a burst mode to eliminate the address setup time and row and column line precharge time after the first access. In burst mode, a series of data bits can be clocked out rapidly after the first bit has been accessed. This mode is useful when all the bits to be accessed are in sequence and in the same row of the array as the initial access. In addition, the SDRAM has a multiple-bank internal architecture that improves opportunities for on-chip parallelism.

The mode register and associated control logic is another key feature differentiating SDRAMs from conventional DRAMs. It provides a mechanism to customize the SDRAM to suit specific system needs. The mode register specifies the burst length, which is the number of separate units of data synchronously fed onto the bus. The register also allows the programmer to adjust the latency between receipt of a read request and the beginning of data transfer.

The SDRAM performs best when it is transferring large blocks of data sequentially, such as for applications like word processing, spreadsheets, and multimedia.

Figure 5.13 shows an example of SDRAM operation. In this case, the burst length is 4 and the latency is 2. The burst read command is initiated by having $\overline{CS}$ and $\overline{CAS}$ low while holding $\overline{RAS}$ and $\overline{WE}$ high at the rising edge of the clock. The address inputs determine the starting column address for the burst, and the mode register sets the type of burst (sequential or interleave) and the burst length (1, 2, 4, 8, full page). The delay from the start of the command to when the data from the first cell appears on the outputs is equal to the value of the $\overline{CAS}$ latency that is set in the mode register.

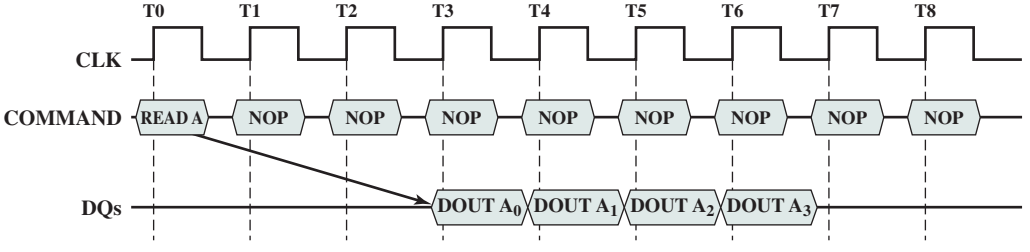


Figure 5.13 SDRAM Read Timing (burst length = 4, $\overline{CAS}$ latency = 2)

DDR SDRAM

Although SDRAM is a significant improvement on asynchronous RAM, it still has shortcomings that unnecessarily limit that I/O data rate that can be achieved. To address these shortcomings a newer version of SDRAM, referred to as double-data-rate DRAM (DDR DRAM) provides several features that dramatically increase the data rate. DDR DRAM was developed by the JEDEC Solid State Technology Association, the Electronic Industries Alliance's semiconductor-engineering-standardization body. Numerous companies make DDR chips, which are widely used in desktop computers and servers.

DDR achieves higher data rates in three ways. First, the data transfer is synchronized to both the rising and falling edge of the clock, rather than just the rising edge. This doubles the data rate; hence the term *double data rate*. Second, DDR uses higher clock rate on the bus to increase the transfer rate. Third, a buffering scheme is used, as explained subsequently.

JEDEC has thus far defined four generations of the DDR technology (Table 5.4). The initial DDR version makes use of a 2-bit prefetch buffer. The prefetch buffer is a memory cache located on the SDRAM chip. It enables the SDRAM chip to preposition bits to be placed on the data bus as rapidly as possible. The DDR I/O bus uses the same clock rate as the memory chip, but because it can handle two bits per cycle, it achieves a data rate that is double the clock rate. The 2-bit prefetch buffer enables the SDRAM chip to keep up with the I/O bus.

To understand the operation of the prefetch buffer, we need to look at it from the point of view of a word transfer. The prefetch buffer size determines how many words of data are fetched (across multiple SDRAM chips) every time a column command is performed with DDR memories. Because the core of the DRAM is much slower than the interface, the difference is bridged by accessing information in parallel and then serializing it out the interface through a multiplexor (MUX). Thus, DDR prefetches two words, which means that every time a read or a write operation is performed, it is performed on two words of data, and bursts out of, or into, the SDRAM over one clock cycle on both clock edges for a total of two consecutive operations. As a result, the DDR I/O interface is twice as fast as the SDRAM core.

Although each new generation of SDRAM results is much greater capacity, the core speed of the SDRAM has not changed significantly from generation to generation. To achieve greater data rates than those afforded by the rather modest increases in SDRAM clock rate, JEDEC increased the buffer size. For DDR2, a 4-bit buffer is used, allowing for words to be transferred in parallel, increasing the effective data rate by a factor of 4. For DDR3, an 8-bit buffer is used and a factor of 8 speedup is achieved (Figure 5.14).

Table 5.4 DDR Characteristics

	DDR1	DDR2	DDR3	DDR4
Prefetch buffer (bits)	2	4	8	8
Voltage level (V)	2.5	1.8	1.5	1.2
Front side bus data rates (Mbps)	200—400	400—1066	800—2133	2133—4266

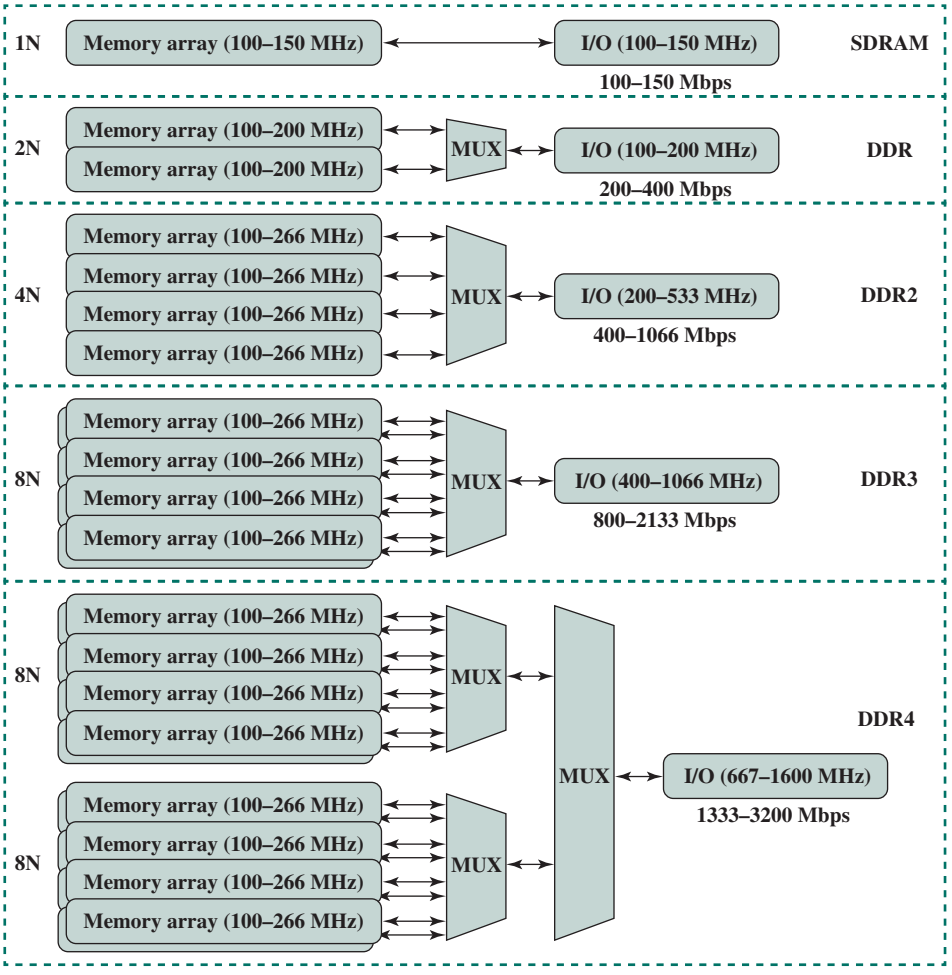


Figure 5.14 DDR Generations

The downside to the prefetch is that it effectively determines the minimum burst length for the SDRAMs. For example, it is very difficult to have an efficient burst length of four words with DDR3's prefetch of eight. Accordingly, the JEDEC designers chose not to increase the buffer size to 16 bits for DDR4, but rather to introduce the concept of a **bank group** [ALLA13]. Bank groups are separate entities such that they allow a column cycle to complete within a bank group, but that column cycle does not impact what is happening in another bank group. Thus, two prefetches of eight can be operating in parallel in the two bank groups. This arrangement keeps the prefetch buffer size the same as for DDR3, while increasing performance as if the prefetch is larger.

Figure 5.14 shows a configuration with two bank groups. With DDR4, up to 4 bank groups can be used.

5.4 FLASH MEMORY

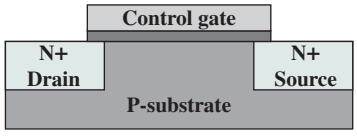
Another form of semiconductor memory is flash memory. Flash memory is used both for internal memory and external memory applications. Here, we provide a technical overview and look at its use for internal memory.

First introduced in the mid-1980s, flash memory is intermediate between EPROM and EEPROM in both cost and functionality. Like EEPROM, flash memory uses an electrical erasing technology. An entire flash memory can be erased in one or a few seconds, which is much faster than EPROM. In addition, it is possible to erase just blocks of memory rather than an entire chip. Flash memory gets its name because the microchip is organized so that a section of memory cells are erased in a single action or "flash." However, flash memory does not provide byte-level erasure. Like EPROM, flash memory uses only one transistor per bit, and so achieves the high density (compared with EEPROM) of EPROM.

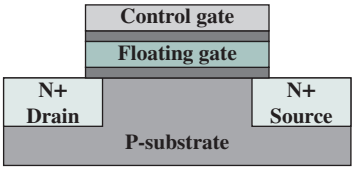
Operation

Figure 5.15 illustrates the basic operation of a flash memory. For comparison, Figure 5.15a depicts the operation of a transistor. Transistors exploit the properties of semiconductors so that a small voltage applied to the gate can be used to control the flow of a large current between the source and the drain.

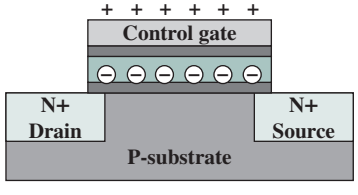
In a flash memory cell, a second gate—called a floating gate, because it is insulated by a thin oxide layer—is added to the transistor. Initially, the floating gate does not interfere with the operation of the transistor (Figure 5.15b). In this state, the cell is deemed to represent binary 1. Applying a large voltage across the oxide layer causes electrons to tunnel through it and become trapped on the floating gate, where they remain even if the power is disconnected (Figure 5.15c). In this state, the cell is deemed to represent binary 0. The state of the cell can be read by using external circuitry to test whether the transistor is working or not. Applying a large voltage in the opposite direction removes the electrons from the floating gate, returning to a state of binary 1.



(a) Transistor structure



(b) Flash memory cell in one state



(c) Flash memory cell in zero state

Figure 5.15 Flash Memory Operation

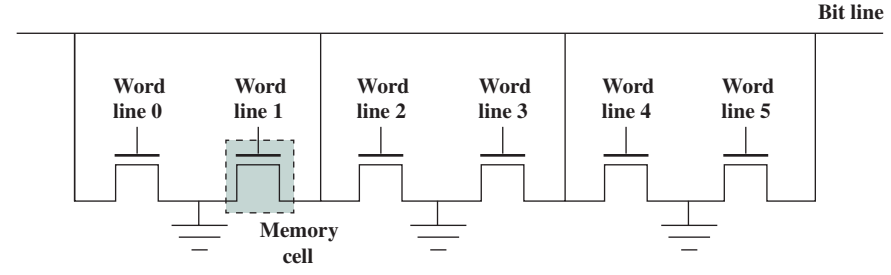
An important characteristic of flash memory is that it is persistent memory, which means that it retains data when there is no power applied to the memory. Thus, it is useful for secondary (external) storage, and as an alternative to random access memory in computers.

NOR and NAND Flash Memory

There are two distinctive types of flash memory, designated as NOR and NAND (Figure 5.16). In **NOR flash memory**, the basic unit of access is a bit, referred to as a *memory cell*. Cells in NOR flash are connected in parallel to the bit lines so that each cell can be read/write/erased individually. If any memory cell of the device is turned on by the corresponding word line, the bit line goes low. This is similar in function to a NOR logic gate.[2]

NAND flash memory is organized in transistor arrays with 16 or 32 transistors in series. The bit line goes low only if all the transistors in the corresponding word lines are turned on. This is similar in function to a NAND logic gate.

Although the specific quantitative values of various characteristics of NOR and NAND are changing year by year, the relative differences between the two types has remained stable. These differences are usefully illustrated by the Kiviat graphs[3] shown in Figure 5.17.



(a) NOR flash structure

(b) NAND flash structure

Figure 5.16 Flash Memory Structures

[2]The circles associated with and in Figure 5.2b indicate signal negation.

[3]A Kiviat graph provides a pictorial means of comparing systems along multiple variables [MORR74]. The variables are laid out at as lines of equal angular intervals within a circle, each line going from the center of the circle to the circumference. A given system is defined by one point on each line; the closer to the circumference, the better the value. The points are connected to yield a shape that is characteristic of that system. The more area enclosed in the shape, the "better" is the system.

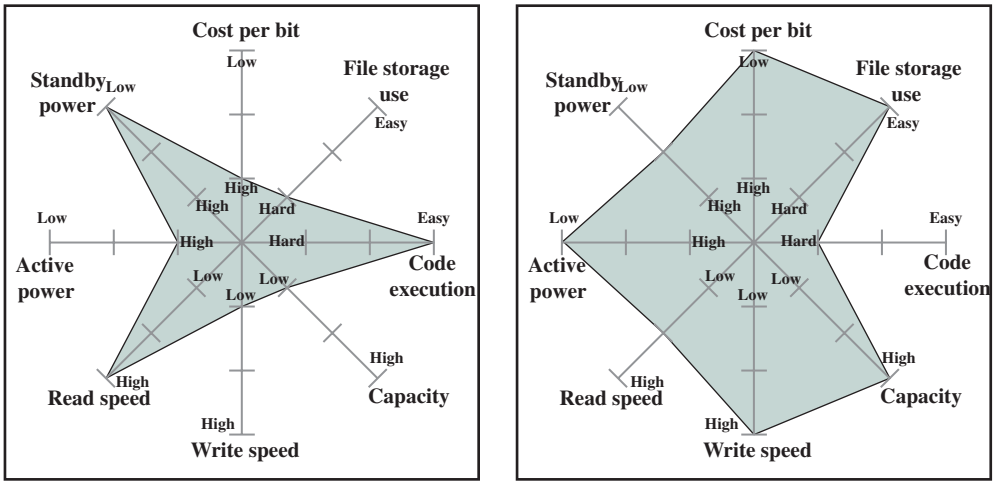


Figure 5.17 Kiviat Graphs for Flash Memory

NOR flash memory provides high-speed random access. It can read and write data to specific locations, and can reference and retrieve a single byte. NAND reads and writes in small blocks. NAND provides higher bit density than NOR and greater write speed. NAND flash does not provide a random-access external address bus so the data must be read on a blockwise basis (also known as page access), where each block holds hundreds to thousands of bits.

For internal memory in embedded systems, NOR flash memory has traditionally been preferred. NAND memory has made some inroads, but NOR remains the dominant technology for internal memory. It is ideally suited for microcontrollers where the amount of program code is relatively small and a certain amount of application data does not vary. For example, the flash memory in Figure 1.16 is NOR memory.

NAND memory is better suited for external memory, such as USB flash drives, memory cards (in digital cameras, MP3 players, etc.), and in what are known as solid-state disks (SSDs). We discuss SSDs in Chapter 6.

5.5 NEWER NONVOLATILE SOLID-STATE MEMORY TECHNOLOGIES

The traditional memory hierarchy has consisted of three levels (Figure 5.18):

- **Static RAM (SRAM):** SRAM provides rapid access time, but is the most expensive and the least dense (bit density). SRAM is suitable for cache memory.
- **Dynamic RAM (DRAM):** Cheaper, denser, and slower than SRAM, DRAM has traditionally been the choice off-chip main memory.
- **Hard disk:** A magnetic disk provides very high bit density and very low cost per bit, with relatively slow access times. It is the traditional choice for external storage as part of the memory hierarchy.

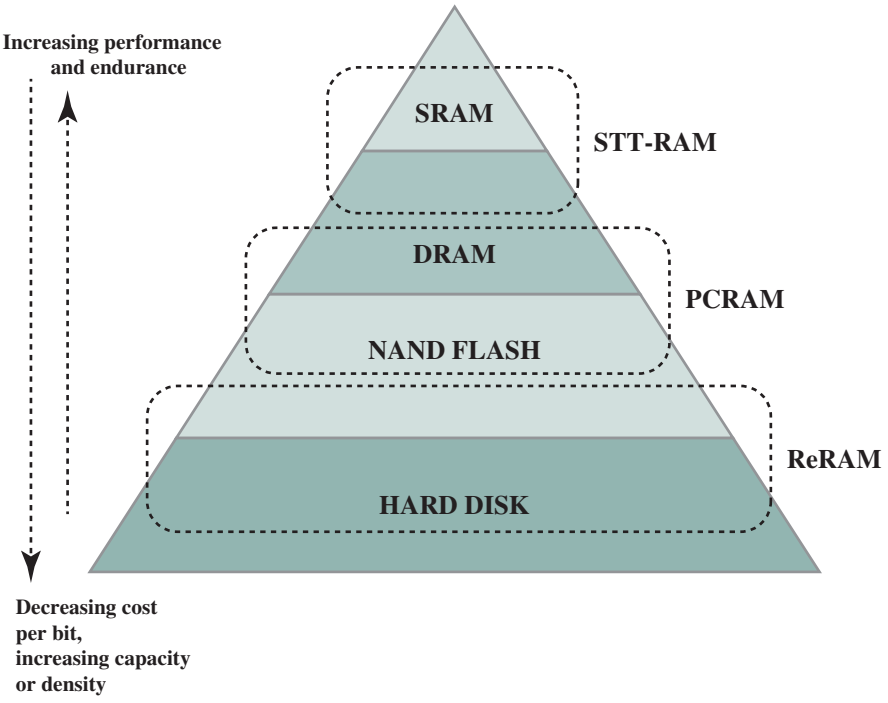


Figure 5.18 Nonvolatile RAM within the Memory Hierarchy

Into this mix, as we have seen, as been added flash memory. Flash memory has the advantage over traditional memory that it is nonvolatile. NOR flash is best suited to storing programs and static application data in embedded systems, while NAND flash has characteristics intermediate between DRAM and hard disks.

Over time, each of these technologies has seen improvements in scaling: higher bit density, higher speed, lower power consumption, and lower cost. However, for semiconductor memory, it is becoming increasingly difficult to continue the pace of improvement [ITRS14].

Recently, there have been breakthroughs in developing new forms of nonvolatile semiconductor memory that continue scaling beyond flash memory. The most promising technologies are spin-transfer torque RAM (STT-RAM), phase-change RAM (PCRAM), and resistive RAM (ReRAM) ([ITRS14], [GOER12]). All of these are in volume production. However, because NAND Flash and to some extent NOR Flash are still dominating the applications, these emerging memories have been used in specialty applications and have not yet fulfilled their original promise to become dominating mainstream high-density nonvolatile memory. This is likely to change in the next few years.

Figure 5.18 shows how these three technologies are likely to fit into the memory hierarchy.

STT-RAM

STT-RAM is a new type of **magnetic RAM (MRAM)**, which features non-volatility, fast writing/reading speed (< 10 ns), and high programming endurance ($> 10^{15}$ cycles) and zero standby power [KULT13]. The storage capability or programmability of MRAM arises from magnetic tunneling junction (MTJ), in which a thin tunneling dielectric is sandwiched between two ferromagnetic layers. One ferromagnetic layer (pinned or reference layer) is designed to have its magnetization pinned, while the magnetization of the other layer (free layer) can be flipped by a write event. An MTJ has a low (high) resistance if the magnetizations of the free layer and the pinned layer are parallel (anti-parallel). In first-generation MRAM design, the magnetization of the free layer is changed by the current-induced magnetic field. In STT-RAM, a new write mechanism, called *polarization-current-induced magnetization switching*, is introduced. For STT-RAM, the magnetization of the free layer is flipped by the electrical current directly. Because the current required to switch an MTJ resistance state is proportional to the MTJ cell area, STT-RAM is believed to have a better scaling property than the first-generation MRAM. Figure 5.19a illustrates the general configuration.

STT-RAM is a good candidate for either cache or main memory.

PCRAM

Phase-change RAM (PCRAM) is the most mature or the new technologies, with an extensive technical literature ([RAOU09], [ZHOU09], [LEE10]).

PCRAM technology is based on a chalcogenide alloy material, which is similar to those commonly used in optical storage media (compact discs and digital versatile discs). The data storage capability is achieved from the resistance differences between an amorphous (high-resistance) and a crystalline (low-resistance) phase of the chalcogenide-based material. In SET operation, the phase change material is crystallized by applying an electrical pulse that heats a significant portion of the cell above its crystallization temperature. In RESET operation, a larger electrical current is applied and then abruptly cut off in order to melt and then quench the material, leaving it in the amorphous state. Figure 5.19b illustrates the general configuration.

PCRAM is a good candidate to replace or supplement DRAM for main memory.

ReRAM

ReRAM (also known as RRAM) works by creating resistance rather than directly storing charge. An electric current is applied to a material, changing the resistance of that material. The resistance state can then be measured and a 1 or 0 is read as the result. Much of the work done on ReRAM to date has focused on finding appropriate materials and measuring the resistance state of the cells. ReRAM designs are low voltage, endurance is far superior to flash memory, and the cells are much smaller—at least in theory. Figure 5.19c shows one ReRam configuration.

ReRAM is a good candidate to replace or supplement both secondary storage and main memory.

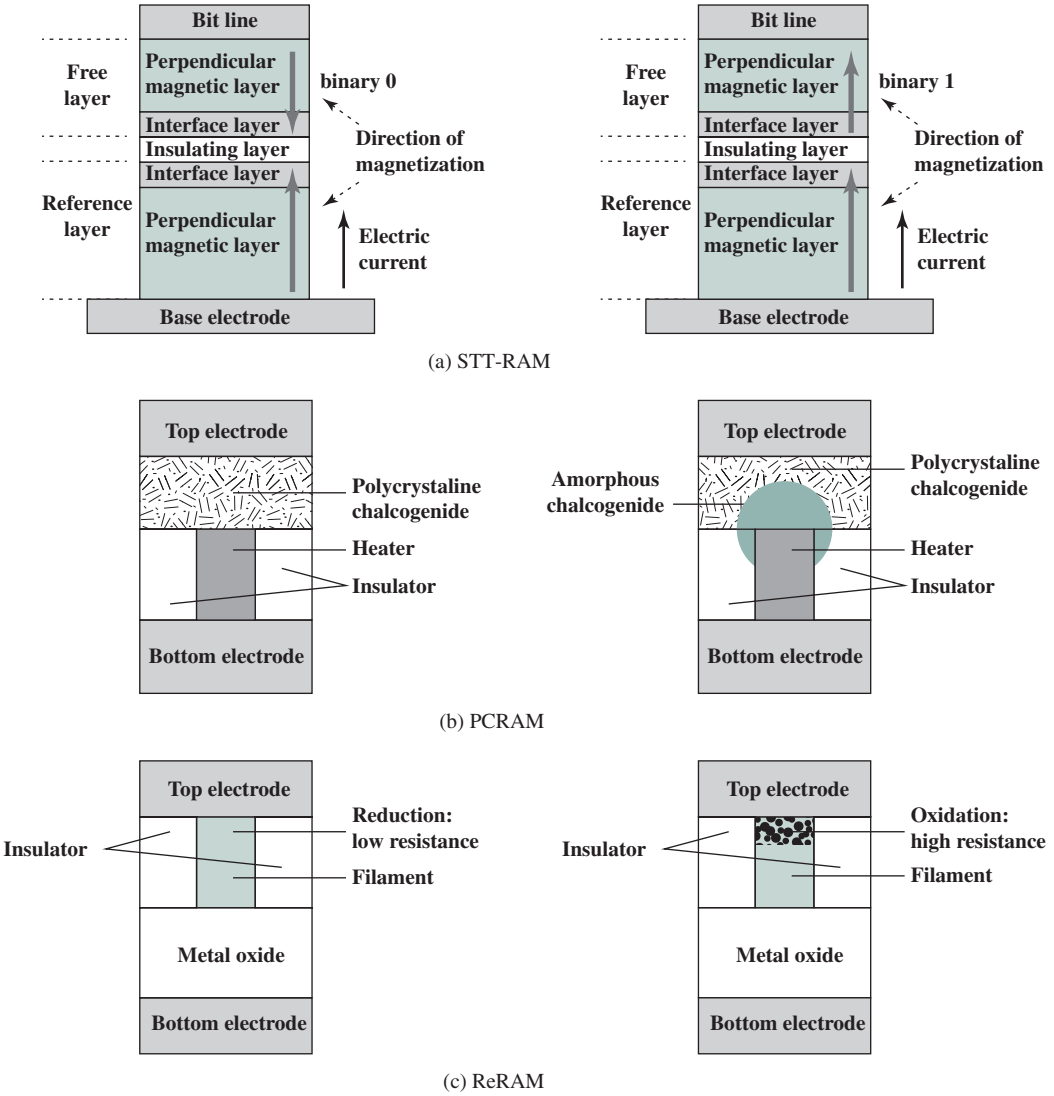


Figure 5.19 Nonvolatile RAM Technologies

5.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

bank group double data rate DRAM (DDR DRAM) dynamic RAM (DRAM)	electrically erasable programmable ROM (EEPROM) erasable programmable ROM (EPROM)	error correcting code (ECC) error correction flash memory Hamming code hard failure

magnetic RAM (MRAM)	read-mostly memory	soft error
NAND flash memory	read-only memory (ROM)	spin-transfer torque RAM (STT-RAM)
nonvolatile memory	resistive RAM (ReRAM)	static RAM (SRAM)
NOR flash memory	semiconductor memory	synchronous DRAM (SDRAM)
phase-change RAM (PCRAM)	single-error-correcting (SEC) code	syndrome
programmable ROM (PROM)	single-error-correcting, double-error-detecting (SEC-DED) code	volatile memory
random access memory (RAM)		

Review Questions

5.1 What are the key properties of semiconductor memory?
5.2 What are two interpretations of the term *random-access memory*?
5.3 What is the difference between DRAM and SRAM in terms of application?
5.4 What is the difference between DRAM and SRAM in terms of characteristics such as speed, size, and cost?
5.5 Explain why one type of RAM is considered to be analog and the other digital.
5.6 What are some applications for ROM?
5.7 What are the differences among EPROM, EEPROM, and flash memory?
5.8 Explain the function of each pin in Figure 5.4b.
5.9 What is a parity bit?
5.10 How is the syndrome for the Hamming code interpreted?
5.11 How does SDRAM differ from ordinary DRAM?
5.12 What is DDR RAM?
5.13 What is the difference between NAND and NOR flash memory?
5.14 List and briefly define three newer nonvolatile solid-state memory technologies.

Problems

5.1 Suggest reasons why RAMs traditionally have been organized as only one bit per chip whereas ROMs are usually organized with multiple bits per chip.
5.2 Consider a dynamic RAM that must be given a refresh cycle 64 times per ms. Each refresh operation requires 150 ns; a memory cycle requires 250 ns. What percentage of the memory's total operating time must be given to refreshes?
5.3 Figure 5.20 shows a simplified timing diagram for a DRAM read operation over a bus. The access time is considered to last from t_1 to t_2. Then there is a recharge time, lasting from t_2 to t_3, during which the DRAM chips will have to recharge before the processor can access them again.
a. Assume that the access time is 60 ns and the recharge time is 40 ns. What is the memory cycle time? What is the maximum data rate this DRAM can sustain, assuming a 1-bit output?
b. Constructing a 32-bit wide memory system using these chips yields what data transfer rate?
5.4 Figure 5.6 indicates how to construct a module of chips that can store 1 MB based on a group of four 256-Kbyte chips. Let's say this module of chips is packaged as a single 1-MB chip, where the word size is 1 byte. Give a high-level chip diagram of how to construct an 8-MB computer memory using eight 1-MB chips. Be sure to show the address lines in your diagram and what the address lines are used for.

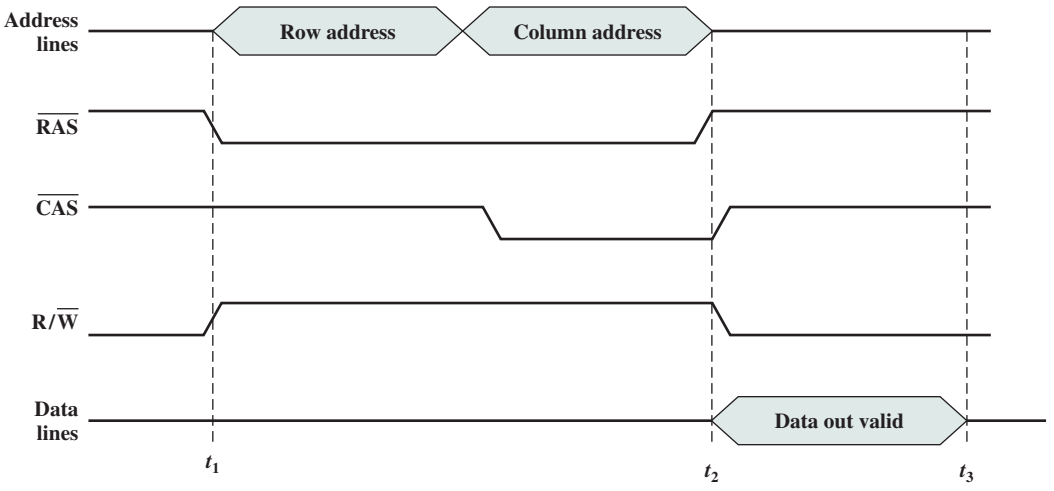


Figure 5.20 Simplified DRAM Read Timing

5.5 On a typical Intel 8086-based system, connected via system bus to DRAM memory, for a read operation, $\overline{\text{RAS}}$ is activated by the trailing edge of the Address Enable signal (Figure C.1 in Appendix C). However, due to propagation and other delays, $\overline{\text{RAS}}$ does not go active until 50 ns after Address Enable returns to a low. Assume the latter occurs in the middle of the second half of state T_1 (somewhat earlier than in Figure C.1). Data are read by the processor at the end of T_3. For timely presentation to the processor, however, data must be provided 60 ns earlier by memory. This interval accounts for propagation delays along the data paths (from memory to processor) and processor data hold time requirements. Assume a clocking rate of 10 MHz.

a. How fast (access time) should the DRAMs be if no wait states are to be inserted?

b. How many wait states do we have to insert per memory read operation if the access time of the DRAMs is 150 ns?

5.6 The memory of a particular microcomputer is built from 64K × 1 DRAMs. According to the data sheet, the cell array of the DRAM is organized into 256 rows. Each row must be refreshed at least once every 4 ms. Suppose we refresh the memory on a strictly periodic basis.

a. What is the time period between successive refresh requests?

b. How long a refresh address counter do we need?

5.7 Figure 5.21 shows one of the early SRAMs, the 16 × 4 Signetics 7489 chip, which stores 16 4-bit words.

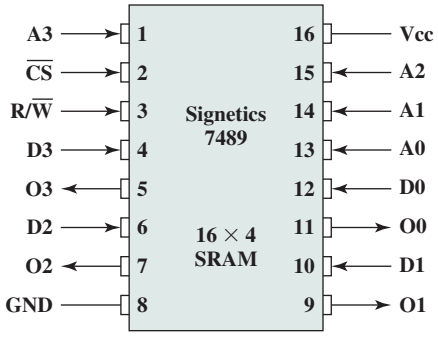
a. List the mode of operation of the chip for each $\overline{\text{CS}}$ input pulse shown in Figure 5.21c.

b. List the memory contents of word locations 0 through 6 after pulse n.

c. What is the state of the output data leads for the input pulses h through m?

5.8 Design a 16-bit memory of total capacity 8192 bits using SRAM chips of size 64 × 1 bit. Give the array configuration of the chips on the memory board showing all required input and output signals for assigning this memory to the lowest address space. The design should allow for both byte and 16-bit word accesses.

5.9 A common unit of measure for failure rates of electronic components is the **Failure unIT** (FIT), expressed as a rate of failures per billion device hours. Another well known but less used measure is **mean time between failures (MTBF),** which is the average time of operation of a particular component until it fails. Consider a 1 MB memory of a 16-bit microprocessor with 256K × 1 DRAMs. Calculate its MTBF assuming 2000 FITS for each DRAM.

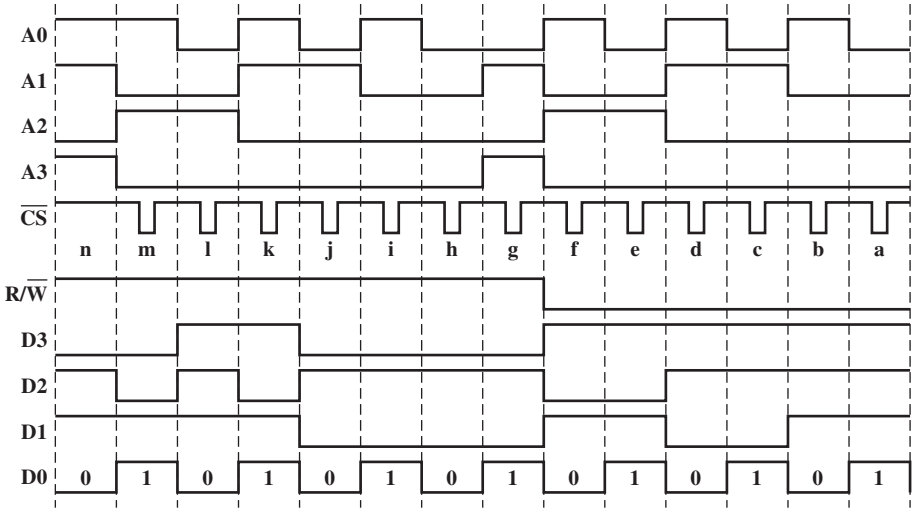


(a) Pin layout

Operating Mode	Inputs			Outputs
	$\overline{CS}$	R/$\overline{W}$	D*n*	O*n*
Write	L	L	L	L
	L	L	H	H
Read	L	H	X	Data
Inhibit writing	H	L	L	H
	H	L	H	L
Store - disable outputs	H	H	X	H

H = high voltage level
L = low voltage level
X = don't care

(b) Truth table



(c) Pulse train

Figure 5.21 The Signetics 7489 SRAM

5.10 For the Hamming code shown in Figure 5.10, show what happens when a check bit rather than a data bit is in error?

5.11 Suppose an 8-bit data word stored in memory is 11000010. Using the Hamming algorithm, determine what check bits would be stored in memory with the data word. Show how you got your answer.

5.12 For the 8-bit word 00111001, the check bits stored with it would be 0111. Suppose when the word is read from memory, the check bits are calculated to be 1101. What is the data word that was read from memory?

5.13 How many check bits are needed if the Hamming error correction code is used to detect single bit errors in a 1024-bit data word?

5.14 Develop an SEC code for a 16-bit data word. Generate the code for the data word 0101000000111001. Show that the code will correctly identify an error in data bit 5.

CHAPTER 6

EXTERNAL MEMORY

6.1 **Magnetic Disk**
- Magnetic Read and Write Mechanisms
- Data Organization and Formatting
- Physical Characteristics
- Disk Performance Parameters

6.2 **RAID**
- RAID Level 0
- RAID Level 1
- RAID Level 2
- RAID Level 3
- RAID Level 4
- RAID Level 5
- RAID Level 6

6.3 **Solid State Drives**
- SSD Compared to HDD
- SSD Organization
- Practical Issues

6.4 **Optical Memory**
- Compact Disk
- Digital Versatile Disk
- High-Definition Optical Disks

6.5 **Magnetic Tape**

6.6 **Key Terms, Review Questions, and Problems**

Learning Objectives

After studying this chapter, you should be able to:

- Understand the key properties of magnetic disks.
- Understand the performance issues involved in **magnetic disk** access.
- Explain the concept of **RAID** and describe the various levels.
- Compare and contrast hard disk drives and solid disk drives.
- Describe in general terms the operation of **flash memory**.
- Understand the differences among the different optical disk storage media.
- Present an overview of **magnetic tape** storage technology.

This chapter examines a range of external memory devices and systems. We begin with the most important device, the magnetic disk. Magnetic disks are the foundation of external memory on virtually all computer systems. The next section examines the use of disk arrays to achieve greater performance, looking specifically at the family of systems known as RAID (Redundant Array of Independent Disks). An increasingly important component of many computer systems is the solid state disk, which is discussed next. Then, external **optical memory** is examined. Finally, magnetic tape is described.

6.1 MAGNETIC DISK

A disk is a circular **platter** constructed of nonmagnetic material, called the **substrate**, coated with a magnetizable material. Traditionally, the substrate has been an aluminum or aluminum alloy material. More recently, glass substrates have been introduced. The glass substrate has a number of benefits, including the following:

- Improvement in the uniformity of the magnetic film surface to increase disk reliability.
- A significant reduction in overall surface defects to help reduce read-write errors.
- Ability to support lower fly heights (described subsequently).
- Better stiffness to reduce disk dynamics.
- Greater ability to withstand shock and damage.

Magnetic Read and Write Mechanisms

Data are recorded on and later retrieved from the disk via a conducting coil named the **head**; in many systems, there are two heads, a read head and a write head. During a read or write operation, the head is stationary while the platter rotates beneath it.

The write mechanism exploits the fact that electricity flowing through a coil produces a magnetic field. Electric pulses are sent to the write head, and the resulting magnetic patterns are recorded on the surface below, with different patterns for positive and negative currents. The write head itself is made of easily magnetizable

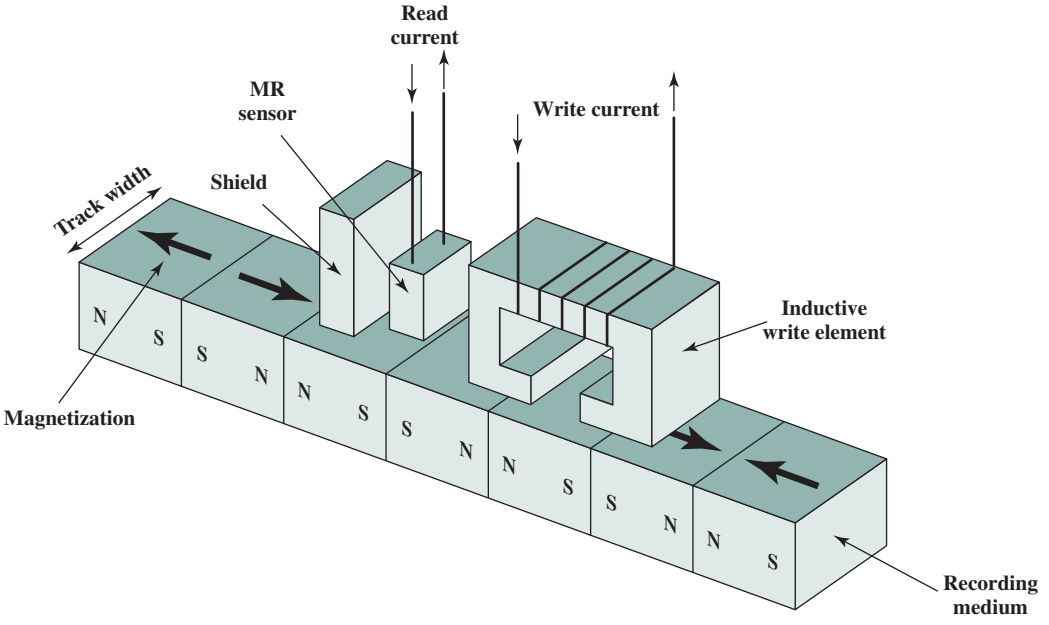


Figure 6.1 Inductive Write/Magnetoresistive Read Head

material and is in the shape of a rectangular doughnut with a gap along one side and a few turns of conducting wire along the opposite side (Figure 6.1). An electric current in the wire induces a magnetic field across the gap, which in turn magnetizes a small area of the recording medium. Reversing the direction of the current reverses the direction of the magnetization on the recording medium.

The traditional read mechanism exploits the fact that a magnetic field moving relative to a coil produces an electrical current in the coil. When the surface of the disk rotates under the head, it generates a current of the same polarity as the one already recorded. The structure of the head for reading is in this case essentially the same as for writing and therefore the same head can be used for both. Such single heads are used in floppy disk systems and in older rigid disk systems.

Contemporary rigid disk systems use a different read mechanism, requiring a separate read head, positioned for convenience close to the write head. The read head consists of a partially shielded **magnetoresistive (MR)** sensor. The MR material has an electrical resistance that depends on the direction of the magnetization of the medium moving under it. By passing a current through the MR sensor, resistance changes are detected as voltage signals. The MR design allows higher-frequency operation, which equates to greater storage densities and operating speeds.

Data Organization and Formatting

The head is a relatively small device capable of reading from or writing to a portion of the platter rotating beneath it. This gives rise to the organization of data on the platter in a concentric set of rings, called **tracks**. Each track is the same width as the head. There are thousands of tracks per surface.

Figure 6.2 depicts this data layout. Adjacent tracks are separated by **intertrack gaps**. This prevents, or at least minimizes, errors due to misalignment of the head or simply interference of magnetic fields. Data are transferred to and from the disk in **sectors**. There are typically hundreds of sectors per track, and these may be of either fixed or variable length. In most contemporary systems, fixed-length sectors are used, with 512 bytes being the nearly universal sector size. To avoid imposing unreasonable precision requirements on the system, adjacent sectors are separated by intersector gaps.

A bit near the center of a rotating disk travels past a fixed point (such as a read–write head) slower than a bit on the outside. Therefore, some way must be found to compensate for the variation in speed so that the head can read all the bits at the same rate. This can be done by defining a variable spacing between bits of information recorded in

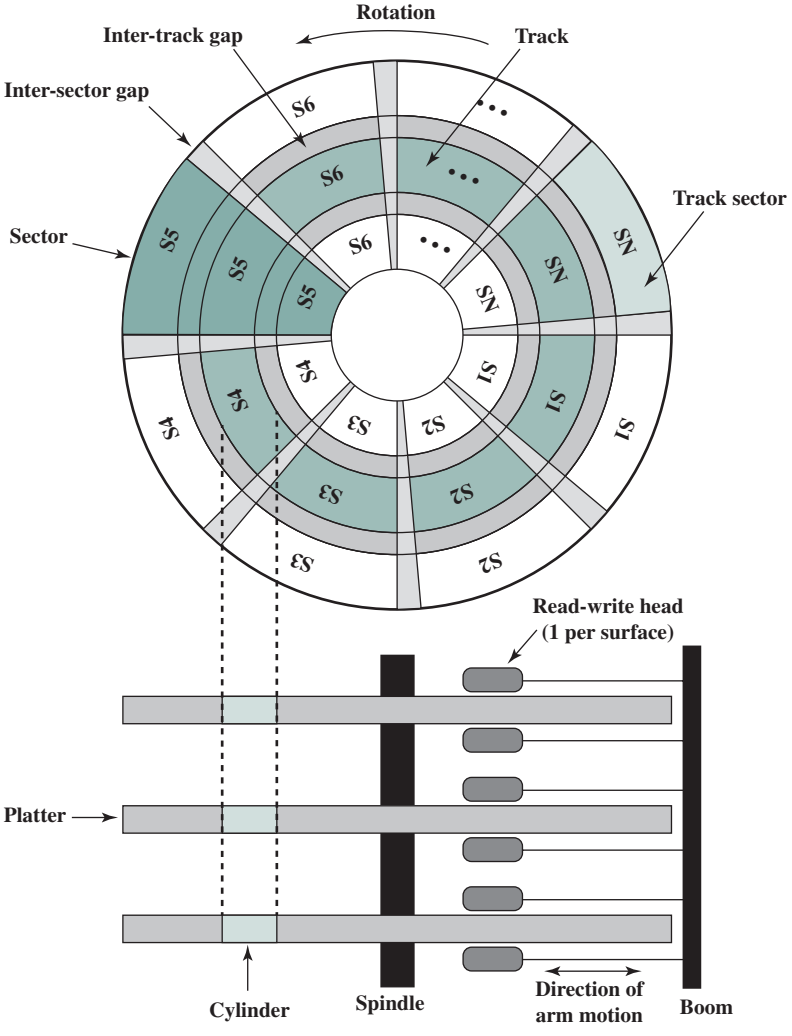
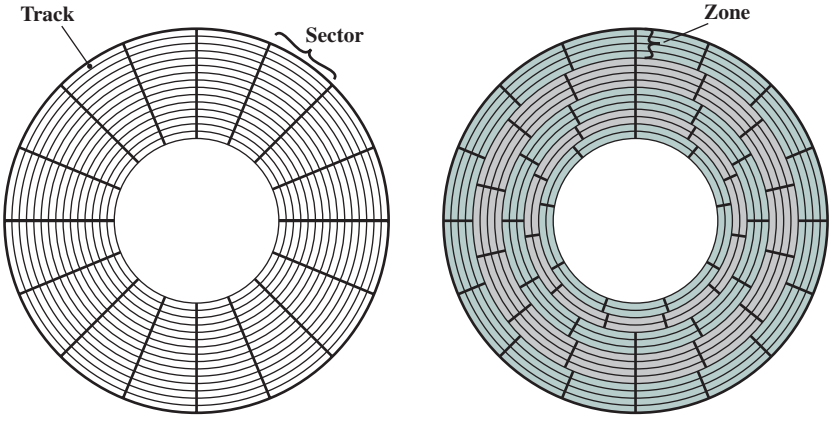


Figure 6.2 Disk Data Layout

locations on the disk, in a way that the outermost tracks has sectors with bigger spacing. The information can then be scanned at the same rate by rotating the disk at a fixed speed, known as the **constant angular velocity (CAV)**. Figure 6.3a shows the layout of a disk using CAV. The disk is divided into a number of pie-shaped sectors and into a series of concentric tracks. The advantage of using CAV is that individual blocks of data can be directly addressed by track and sector. To move the head from its current location to a specific address, it only takes a short movement of the head to a specific track and a short wait for the proper sector to spin under the head. The disadvantage of CAV is that the amount of data that can be stored on the long outer tracks is the only same as what can be stored on the short inner tracks.

Because the **density**, in bits per linear inch, increases in moving from the outermost track to the innermost track, disk storage capacity in a straightforward CAV system is limited by the maximum recording density that can be achieved on the innermost track. To maximize storage capacity, it would be preferable to have the same linear bit density on each track. This would require unacceptably complex circuitry. Modern hard disk systems use simpler technique, which approximates equal bit density per track, known as multiple zone recording (MZR), in which the surface is divided into a number of concentric zones (16 is typical). Each zone contains a number of contiguous tracks, typically in the thousands. Within a zone, the number of bits per track is constant. Zones farther from the center contain more bits (more sectors) than zones closer to the center. Zones are defined in such a way that the linear bit density is approximately the same on all tracks of the disk. MZR allows for greater overall storage capacity at the expense of somewhat more complex circuitry. As the disk head moves from one zone to another, the length (along the track) of individual bits changes, causing a change in the timing for reads and writes.

Figure 6.3b is a simplified MZR layout, with 15 tracks organized into 5 zones. The innermost two zones have two tracks each, with each track having nine sectors; the next zone has 3 tracks, each with 12 sectors; and the outermost 2 zones have 4 tracks each, with each track having 16 sectors.



(a) Constant angular velocity (b) Multiple zone recording

Figure 6.3 Comparison of Disk Layout Methods

Some means is needed to locate sector positions within a track. Clearly, there must be some starting point on the track and a way of identifying the start and end of each sector. These requirements are handled by means of control data recorded on the disk. Thus, the disk is formatted with some extra data used only by the disk drive and not accessible to the user.

An example of disk formatting is shown in Figure 6.4. In this case, each track contains 30 fixed-length sectors of 600 bytes each. Each sector holds 512 bytes of data plus control information useful to the disk controller. The ID field is a unique identifier or address used to locate a particular sector. The SYNCH byte is a special bit pattern that delimits the beginning of the field. The track number identifies a track on a surface. The head number identifies a head, because this disk has multiple surfaces (explained presently). The ID and data fields each contain an error-detecting code.

Physical Characteristics

Table 6.1 lists the major characteristics that differentiate among the various types of magnetic disks. First, the head may either be fixed or movable with respect to the radial direction of the platter. In a **fixed-head disk**, there is one read-write head per track. All of the heads are mounted on a rigid arm that extends across all tracks; such systems are rare today. In a **movable-head disk**, there is only one read-write head. Again, the head is mounted on an arm. Because the head must be able to be positioned above any track, the arm can be extended or retracted for this purpose.

The disk itself is mounted in a disk drive, which consists of the arm, a spindle that rotates the disk, and the electronics needed for input and output of binary data. A **nonremovable disk** is permanently mounted in the disk drive; the hard disk in a personal computer is a nonremovable disk. A **removable disk** can be removed and replaced with another disk. The advantage of the latter type is that unlimited amounts of data are available with a limited number of disk systems. Furthermore, such a disk may be moved from one computer system to another. Floppy disks and ZIP cartridge disks are examples of removable disks.

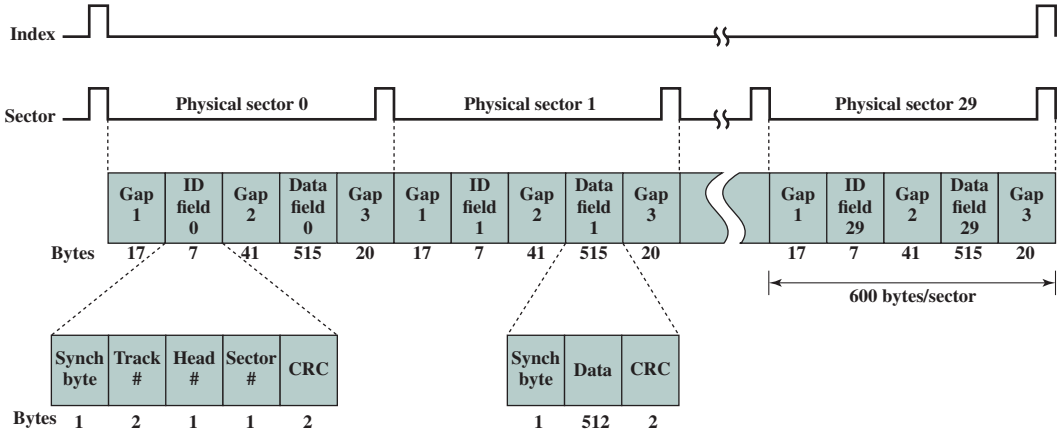


Figure 6.4 Winchester Disk Format (Seagate ST506)

Table 6.1 Physical Characteristics of Disk Systems

Head Motion	**Platters**
Fixed head (one per track)	Single platter
Movable head (one per surface)	Multiple platter
Disk Portability	**Head Mechanism**
Nonremovable disk	Contact (floppy)
Removable disk	Fixed gap
Sides	Aerodynamic gap (Winchester)
Single sided	
Double sided	

For most disks, the magnetizable coating is applied to both sides of the platter, which is then referred to as **double sided**. Some less expensive disk systems use **single-sided** disks.

Some disk drives accommodate **multiple platters** stacked vertically a fraction of an inch apart. Multiple arms are provided (Figure 6.2). Multiple–platter disks employ a movable head, with one read-write head per platter surface. All of the heads are mechanically fixed so that all are at the same distance from the center of the disk and move together. Thus, at any time, all of the heads are positioned over tracks that are of equal distance from the center of the disk. The set of all the tracks in the same relative position on the platter is referred to as a **cylinder**. This is illustrated in Figure 6.2.

Finally, the head mechanism provides a classification of disks into three types. Traditionally, the read-write head has been positioned a fixed distance above the platter, allowing an air gap. At the other extreme is a head mechanism that actually comes into physical contact with the medium during a read or write operation. This mechanism is used with the **floppy disk**, which is a small, flexible platter and the least expensive type of disk.

To understand the third type of disk, we need to comment on the relationship between data density and the size of the air gap. The head must generate or sense an electromagnetic field of sufficient magnitude to write and read properly. The narrower the head is, the closer it must be to the platter surface to function. A narrower head means narrower tracks and therefore greater data density, which is desirable. However, the closer the head is to the disk, the greater the risk of error from impurities or imperfections. To push the technology further, the Winchester disk was developed. Winchester heads are used in sealed drive assemblies that are almost free of contaminants. They are designed to operate closer to the disk's surface than conventional rigid disk heads, thus allowing greater data density. The head is actually an aerodynamic foil that rests lightly on the platter's surface when the disk is motionless. The air pressure generated by a spinning disk is enough to make the foil rise above the surface. The resulting noncontact system can be engineered to use narrower heads that operate closer to the platter's surface than conventional rigid disk heads.

Table 6.2 gives disk parameters for typical contemporary high-performance disks.

Table 6.2 Typical Hard Disk Drive Parameters

Characteristics	Seagate Enterprise	Seagate Barracuda XT	Seagate Cheetah NS	Seagate Laptop HDD
Application	Enterprise	Desktop	Network-attached storage, application servers	Laptop
Capacity	6 TB	3 TB	600 GB	2 TB
Average seek time	4.16 ms	N/A	3.9 ms read 4.2 ms write	13 ms
Spindle speed	7200 rpm	7200 rpm	10,075 rpm	5400 rpm
Average latency	4.16 ms	4.16 ms	2.98	5.6 ms
Maximum sustained transfer rate	216 MB/sec	149 MB/sec	97 MB/sec	300 MB/sec
Bytes per sector	512/4096	512	512	4096
Tracks per cylinder (number of platter surfaces)	8	10	8	4
Cache	128 MB	64 MB	16 MB	8 MB

Disk Performance Parameters

The actual details of disk I/O operation depend on the computer system, the operating system, and the nature of the I/O channel and disk controller hardware. A general timing diagram of disk I/O transfer is shown in Figure 6.5.

When the disk drive is operating, the disk is rotating at constant speed. To read or write, the head must be positioned at the desired track and at the beginning of the desired sector on that track. Track selection involves moving the head in a movable-head system or electronically selecting one head on a fixed-head system. On a movable-head system, the time it takes to position the head at the track is known as **seek time**. In either case, once the track is selected, the disk controller waits until the appropriate sector rotates to line up with the head. The time it takes for the beginning of the sector to reach the head is known as **rotational delay**, or *rotational latency*. The sum of the seek time, if any, and the rotational delay equals the **access time**, which is the time it takes to get into position to read or write. Once the head is in position, the read or write operation is then performed as the sector moves under the head; this is the data transfer portion of the operation; the time required for the transfer is the **transfer time**.

In addition to the access time and transfer time, there are several queuing delays normally associated with a disk I/O operation. When a process issues an I/O

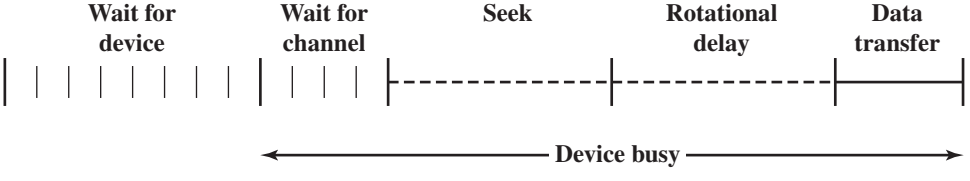


Figure 6.5 Timing of a Disk I/O Transfer

request, it must first wait in a queue for the device to be available. At that time, the device is assigned to the process. If the device shares a single I/O channel or a set of I/O channels with other disk drives, then there may be an additional wait for the channel to be available. At that point, the seek is performed to begin disk access.

In some high-end systems for servers, a technique known as rotational positional sensing (RPS) is used. This works as follows: When the seek command has been issued, the channel is released to handle other I/O operations. When the seek is completed, the device determines when the data will rotate under the head. As that sector approaches the head, the device tries to reestablish the communication path back to the host. If either the control unit or the channel is busy with another I/O, then the reconnection attempt fails and the device must rotate one whole revolution before it can attempt to reconnect, which is called an RPS miss. This is an extra delay element that must be added to the timeline of Figure 6.5.

SEEK TIME Seek time is the time required to move the disk arm to the required track. It turns out that this is a difficult quantity to pin down. The seek time consists of two key components: the initial startup time, and the time taken to traverse the tracks that have to be crossed once the access arm is up to speed. Unfortunately, the traversal time is not a linear function of the number of tracks, but includes a settling time (time after positioning the head over the target track until track identification is confirmed).

Much improvement comes from smaller and lighter disk components. Some years ago, a typical disk was 14 inches (36 cm) in diameter, whereas the most common size today is 3.5 inches (8.9 cm), reducing the distance that the arm has to travel. A typical average seek time on contemporary hard disks is under 10 ms.

ROTATIONAL DELAY Disks, other than floppy disks, rotate at speeds ranging from 3600 rpm (for handheld devices such as digital cameras) up to, as of this writing, 20,000 rpm; at this latter speed, there is one revolution per 3 ms. Thus, on the average, the rotational delay will be 1.5 ms.

TRANSFER TIME The transfer time to or from the disk depends on the rotation speed of the disk in the following fashion:

$$T = \frac{b}{rN}$$

where

T = transfer time
b = number of bytes to be transferred
N = number of bytes on a track
r = rotation speed, in revolutions per second

Thus the total average read or write time T_{total} can be expressed as

$$T_{total} = T_s + \frac{1}{2r} + \frac{b}{rN} \tag{6.1}$$

where T_s is the average seek time. Note that on a zoned drive, the number of bytes per track is variable, complicating the calculation.[1]

[1]Compare the two preceding equations to Equation (4.1).

A TIMING COMPARISON With the foregoing parameters defined, let us look at two different I/O operations that illustrate the danger of relying on average values. Consider a disk with an advertised average seek time of 4 ms, rotation speed of 15,000 rpm, and 512-byte sectors with 500 sectors per track. Suppose that we wish to read a file consisting of 2500 sectors for a total of 1.28 Mbytes. We would like to estimate the total time for the transfer.

First, let us assume that the file is stored as compactly as possible on the disk. That is, the file occupies all of the sectors on 5 adjacent tracks (5 tracks × 500 sectors/track = 2500 sectors). This is known as *sequential organization*. Now, the time to read the first track is as follows:

Average seek	4 ms
Average rotational delay	2 ms
Read 500 sectors	4 ms
	10 ms

Suppose that the remaining tracks can now be read with essentially no seek time. That is, the I/O operation can keep up with the flow from the disk. Then, at most, we need to deal with rotational delay for the four remaining tracks. Thus each successive track is read in 2 + 4 = 6 ms. To read the entire file,

$$\text{Total time} = 10 + (4 \times 6) = 34 \text{ ms} = 0.034 \text{ seconds}$$

Now let us calculate the time required to read the same data using random access rather than sequential access; that is, accesses to the sectors are distributed randomly over the disk. For each sector, we have

Average seek	4 ms
Rotational delay	2 ms
Read 1 sectors	0.008 ms
	6.008 ms

$$\text{Total time} = 2500 \times 6.008 = 15{,}020 \text{ ms} = 15.02 \text{ seconds}$$

It is clear that the order in which sectors are read from the disk has a tremendous effect on I/O performance. In the case of file access in which multiple sectors are read or written, we have some control over the way in which sectors of data are deployed. However, even in the case of a file access, in a multiprogramming environment, there will be I/O requests competing for the same disk. Thus, it is worthwhile to examine ways in which the performance of disk I/O can be improved over that achieved with purely random access to the disk. This leads to a consideration of disk scheduling algorithms, which is the province of the operating system and beyond the scope of this book (see [STAL15] for a discussion).

RAID Simulator

6.2 RAID

As discussed earlier, the rate in improvement in secondary storage performance has been considerably less than the rate for processors and main memory. This mismatch has made the disk storage system perhaps the main focus of concern in improving overall computer system performance.

As in other areas of computer performance, disk storage designers recognize that if one component can only be pushed so far, additional gains in performance are to be had by using multiple parallel components. In the case of disk storage, this leads to the development of arrays of disks that operate independently and in parallel. With multiple disks, separate I/O requests can be handled in parallel, as long as the data required reside on separate disks. Further, a single I/O request can be executed in parallel if the block of data to be accessed is distributed across multiple disks.

With the use of multiple disks, there is a wide variety of ways in which the data can be organized and in which redundancy can be added to improve reliability. This could make it difficult to develop database schemes that are usable on a number of platforms and operating systems. Fortunately, industry has agreed on a standardized scheme for multiple-disk database design, known as RAID (Redundant Array of Independent Disks). The RAID scheme consists of seven levels,[2] zero through six. These levels do not imply a hierarchical relationship but designate different design architectures that share three common characteristics:

1. RAID is a set of physical disk drives viewed by the operating system as a single logical drive.
2. Data are distributed across the physical drives of an array in a scheme known as striping, described subsequently.
3. Redundant disk capacity is used to store parity information, which guarantees data recoverability in case of a disk failure.

The details of the second and third characteristics differ for the different RAID levels. RAID 0 and RAID 1 do not support the third characteristic.

The term *RAID* was originally coined in a paper by a group of researchers at the University of California at Berkeley [PATT88].[3] The paper outlined various RAID configurations and applications and introduced the definitions of the RAID levels that are still used. The RAID strategy employs multiple disk drives and distributes data in such a way as to enable simultaneous access to data from multiple drives, thereby improving I/O performance and allowing easier incremental increases in capacity.

---

[2]Additional levels have been defined by some researchers and some companies, but the seven levels described in this section are the ones universally agreed on.

[3]In that paper, the acronym RAID stood for Redundant Array of Inexpensive Disks. The term *inexpensive* was used to contrast the small relatively inexpensive disks in the RAID array to the alternative, a single large expensive disk (SLED). The SLED is essentially a thing of the past, with similar disk technology being used for both RAID and non-RAID configurations. Accordingly, the industry has adopted the term *independent* to emphasize that the RAID array creates significant performance and reliability gains.

The unique contribution of the RAID proposal is to address effectively the need for redundancy. Although allowing multiple heads and actuators to operate simultaneously achieves higher I/O and transfer rates, the use of multiple devices increases the probability of failure. To compensate for this decreased reliability, RAID makes use of stored parity information that enables the recovery of data lost due to a disk failure.

We now examine each of the RAID levels. Table 6.3 provides a rough guide to the seven levels. In the table, I/O performance is shown both in terms of data transfer capacity, or ability to move data, and I/O request rate, or ability to satisfy I/O requests, since these RAID levels inherently perform differently relative to these two metrics. Each RAID level's strong point is highlighted by darker shading. Figure 6.6 illustrates the use of the seven RAID schemes to support a data capacity requiring four disks with no redundancy. The figures highlight the layout of user data and redundant data and indicates the relative storage requirements of the various levels. We refer to these figures throughout the following discussion. Of the seven RAID levels described, only four are commonly used: RAID levels 0, 1, 5, and 6.

RAID Level 0

RAID level 0 is not a true member of the RAID family because it does not include redundancy to improve performance. However, there are a few applications, such as some on supercomputers in which performance and capacity are primary concerns and low cost is more important than improved reliability.

For RAID 0, the user and system data are distributed across all of the disks in the array. This has a notable advantage over the use of a single large disk: If two-different I/O requests are pending for two different blocks of data, then there is a good chance that the requested blocks are on different disks. Thus, the two requests can be issued in parallel, reducing the I/O queuing time.

But RAID 0, as with all of the RAID levels, goes further than simply distributing the data across a disk array: The data are *striped* across the available disks. This is best understood by considering Figure 6.7. All of the user and system data are viewed as being stored on a logical disk. The logical disk is divided into strips; these strips may be physical blocks, sectors, or some other unit. The strips are mapped round robin to consecutive physical disks in the RAID array. A set of logically consecutive strips that maps exactly one strip to each array member is referred to as a **stripe.** In an n-disk array, the first n logical strips are physically stored as the first strip on each of the n disks, forming the first stripe; the second n strips are distributed as the second strips on each disk; and so on. The advantage of this layout is that if a single I/O request consists of multiple logically contiguous strips, then up to n strips for that request can be handled in parallel, greatly reducing the I/O transfer time.

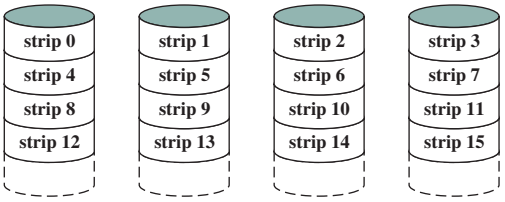
Figure 6.7 indicates the use of array management software to map between logical and physical disk space. This software may execute either in the disk subsystem or in a host computer.

RAID 0 FOR HIGH DATA TRANSFER CAPACITY The performance of any of the RAID levels depends critically on the request patterns of the host system and on the layout of the data. These issues can be most clearly addressed in RAID 0, where the

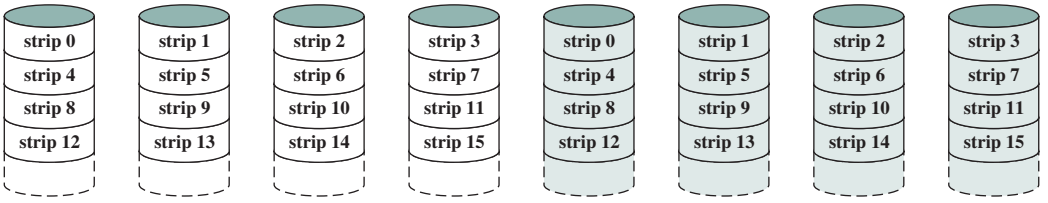
Table 6.3 RAID Levels

Category	Level	Description	Disks Required	Data Availability	Large I/O Data Transfer Capacity	Small I/O Request Rate
Striping	0	Nonredundant	N	Lower than single disk	Very high	Very high for both read and write
Mirroring	1	Mirrored	$2N$	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
Parallel access	2	Redundant via Hamming code	$N + m$	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	3	Bit-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Independent access	4	Block-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
	6	Block-interleaved dual distributed parity	$N + 2$	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

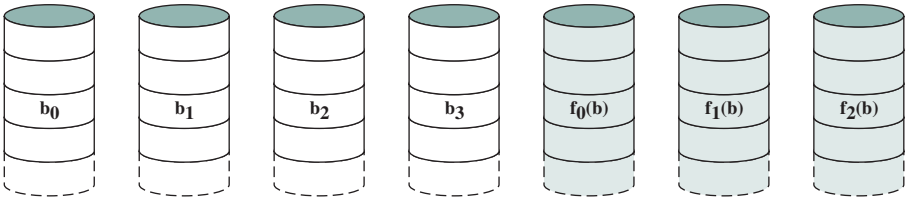
Note: N = number of data disks; m proportional to log N



(a) RAID 0 (Nonredundant)



(b) RAID 1 (Mirrored)



(c) RAID 2 (Redundancy through Hamming code)

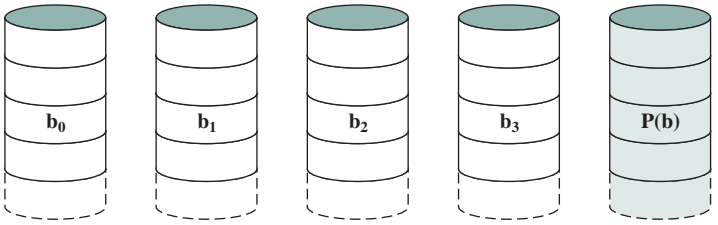
Figure 6.6 RAID Levels (*Continued*)

impact of redundancy does not interfere with the analysis. First, let us consider the use of RAID 0 to achieve a high data transfer rate. For applications to experience a high transfer rate, two requirements must be met. First, a high transfer capacity must exist along the entire path between host memory and the individual disk drives. This includes internal controller buses, host system I/O buses, I/O adapters, and host memory buses.

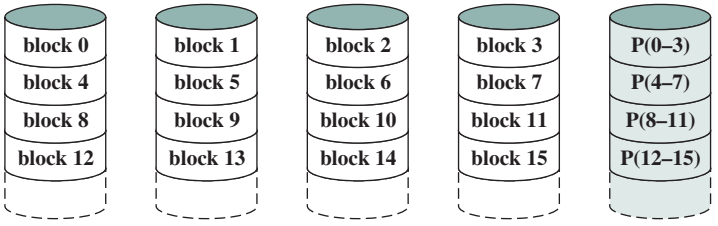
The second requirement is that the application must make I/O requests that drive the disk array efficiently. This requirement is met if the typical request is for large amounts of logically contiguous data, compared to the size of a strip. In this case, a single I/O request involves the parallel transfer of data from multiple disks, increasing the effective transfer rate compared to a single-disk transfer.

RAID 0 FOR HIGH I/O REQUEST RATE In a transaction-oriented environment, the user is typically more concerned with response time than with transfer rate. For an individual I/O request for a small amount of data, the I/O time is dominated by the motion of the disk heads (seek time) and the movement of the disk (rotational latency).

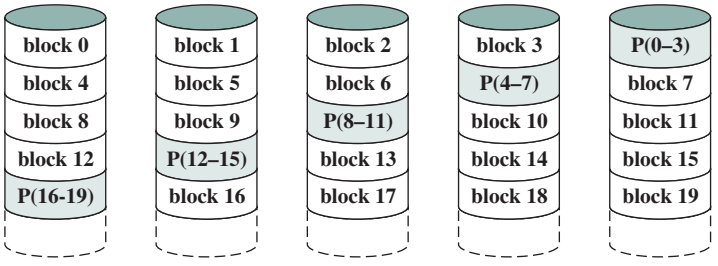
In a transaction environment, there may be hundreds of I/O requests per second. A disk array can provide high I/O execution rates by balancing the I/O load across multiple disks. Effective load balancing is achieved only if there are typically



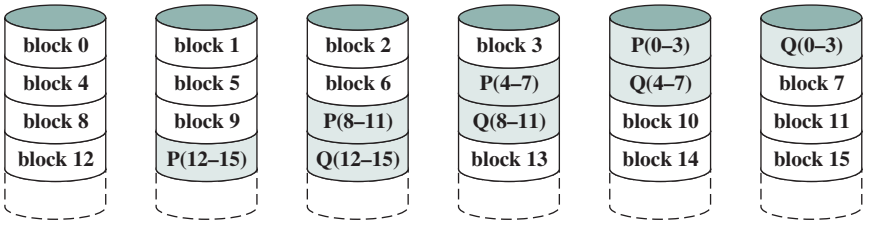
(d) RAID 3 (Bit-interleaved parity)



(e) RAID 4 (Block-level parity)



(f) RAID 5 (Block-level distributed parity)



(g) RAID 6 (Dual redundancy)

Figure 6.6 RAID Levels (*Continued*)

multiple I/O requests outstanding. This, in turn, implies that there are multiple independent applications or a single transaction-oriented application that is capable of multiple asynchronous I/O requests. The performance will also be influenced by the strip size. If the strip size is relatively large, so that a single I/O request only involves a single disk access, then multiple waiting I/O requests can be handled in parallel, reducing the queuing time for each request.

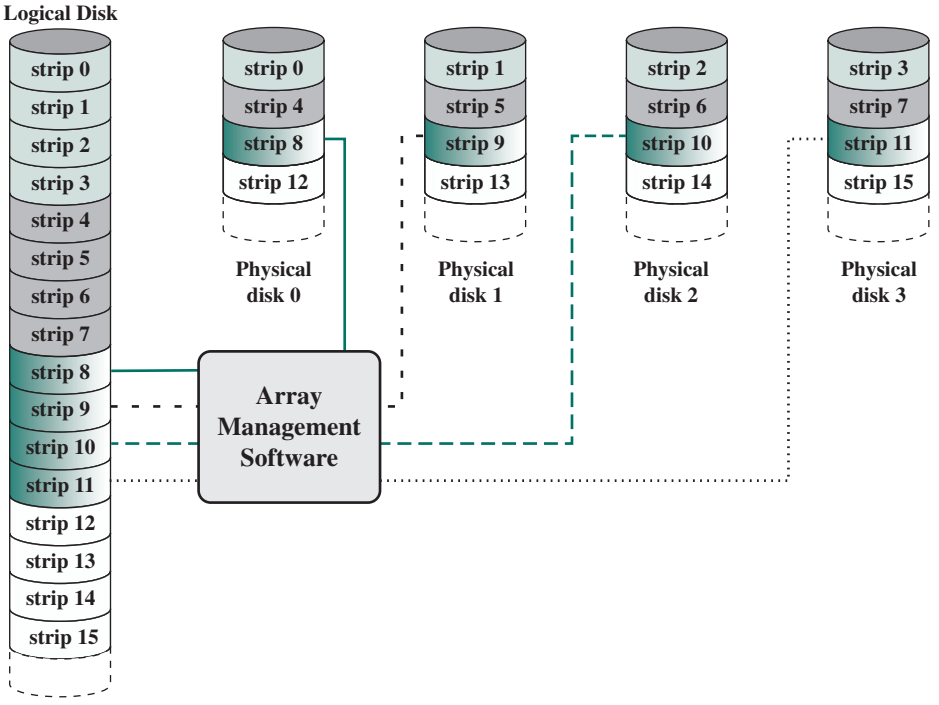


Figure 6.7 Data Mapping for a RAID Level 0 Array

RAID Level 1

RAID 1 differs from RAID levels 2 through 6 in the way in which redundancy is achieved. In these other RAID schemes, some form of parity calculation is used to introduce redundancy, whereas in RAID 1, redundancy is achieved by the simple expedient of duplicating all the data. As Figure 6.6b shows, data striping is used, as in RAID 0. But in this case, each logical strip is mapped to two separate physical disks so that every disk in the array has a mirror disk that contains the same data. RAID 1 can also be implemented without data striping, though this is less common.

There are a number of positive aspects to the RAID 1 organization:

1. A read request can be serviced by either of the two disks that contains the requested data, whichever one involves the minimum seek time plus rotational latency.
2. A write request requires that both corresponding strips be updated, but this can be done in parallel. Thus, the write performance is dictated by the slower of the two writes (i.e., the one that involves the larger seek time plus rotational latency). However, there is no "write penalty" with RAID 1. RAID levels 2 through 6 involve the use of parity bits. Therefore, when a single strip is updated, the array management software must first compute and update the parity bits as well as updating the actual strip in question.
3. Recovery from a failure is simple. When a drive fails, the data may still be accessed from the second drive.

The principal disadvantage of RAID 1 is the cost; it requires twice the disk space of the logical disk that it supports. Because of that, a RAID 1 configuration is likely to be limited to drives that store system software and data and other highly critical files. In these cases, RAID 1 provides real-time copy of all data so that in the event of a disk failure, all of the critical data are still immediately available.

In a transaction-oriented environment, RAID 1 can achieve high I/O request rates if the bulk of the requests are reads. In this situation, the performance of RAID 1 can approach double of that of RAID 0. However, if a substantial fraction of the I/O requests are write requests, then there may be no significant performance gain over RAID 0. RAID 1 may also provide improved performance over RAID 0 for data transfer intensive applications with a high percentage of reads. Improvement occurs if the application can split each read request so that both disk members participate.

RAID Level 2

RAID levels 2 and 3 make use of a parallel access technique. In a parallel access array, all member disks participate in the execution of every I/O request. Typically, the spindles of the individual drives are synchronized so that each disk head is in the same position on each disk at any given time.

As in the other RAID schemes, data striping is used. In the case of RAID 2 and 3, the strips are very small, often as small as a single byte or word. With RAID 2, an error-correcting code is calculated across corresponding bits on each data disk, and the bits of the code are stored in the corresponding bit positions on multiple parity disks. Typically, a Hamming code is used, which is able to correct single-bit errors and detect double-bit errors.

Although RAID 2 requires fewer disks than RAID 1, it is still rather costly. The number of redundant disks is proportional to the log of the number of data disks. On a single read, all disks are simultaneously accessed. The requested data and the associated error-correcting code are delivered to the array controller. If there is a single-bit error, the controller can recognize and correct the error instantly, so that the read access time is not slowed. On a single write, all data disks and parity disks must be accessed for the write operation.

RAID 2 would only be an effective choice in an environment in which many disk errors occur. Given the high reliability of individual disks and disk drives, RAID 2 is overkill and is not implemented.

RAID Level 3

RAID 3 is organized in a similar fashion to RAID 2. The difference is that RAID 3 requires only a single redundant disk, no matter how large the disk array. RAID 3 employs parallel access, with data distributed in small strips. Instead of an error-correcting code, a simple parity bit is computed for the set of individual bits in the same position on all of the data disks.

REDUNDANCY In the event of a drive failure, the parity drive is accessed and data is reconstructed from the remaining devices. Once the failed drive is replaced, the missing data can be restored on the new drive and operation resumed.

Data reconstruction is simple. Consider an array of five drives in which X0 through X3 contain data and X4 is the parity disk. The parity for the *i*th bit is calculated as follows:

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i)$$

where $\oplus$ is exclusive-OR function.

Suppose that drive X1 has failed. If we add $X4(i) \oplus X1(i)$ to both sides of the preceding equation, we get

$$X1(i) = X4(i) \oplus X3(i) \oplus X2(i) \oplus X0(i)$$

Thus, the contents of each strip of data on X1 can be regenerated from the contents of the corresponding strips on the remaining disks in the array. This principle is true for RAID levels 3 through 6.

In the event of a disk failure, all of the data are still available in what is referred to as reduced mode. In this mode, for reads, the missing data are regenerated on the fly using the exclusive-OR calculation. When data are written to a reduced RAID 3 array, consistency of the parity must be maintained for later regeneration. Return to full operation requires that the failed disk be replaced and the entire contents of the failed disk be regenerated on the new disk.

PERFORMANCE Because data are striped in very small strips, RAID 3 can achieve very high data transfer rates. Any I/O request will involve the parallel transfer of data from all of the data disks. For large transfers, the performance improvement is especially noticeable. On the other hand, only one I/O request can be executed at a time. Thus, in a transaction-oriented environment, performance suffers.

RAID Level 4

RAID levels 4 through 6 make use of an independent access technique. In an independent access array, each member disk operates independently, so that separate I/O requests can be satisfied in parallel. Because of this, independent access arrays are more suitable for applications that require high I/O request rates and are relatively less suited for applications that require high data transfer rates.

As in the other RAID schemes, data striping is used. In the case of RAID 4 through 6, the strips are relatively large. With RAID 4, a bit-by-bit parity strip is calculated across corresponding strips on each data disk, and the parity bits are stored in the corresponding strip on the parity disk.

RAID 4 involves a write penalty when an I/O write request of small size is performed. Each time that a write occurs, the array management software must update not only the user data but also the corresponding parity bits. Consider an array of five drives in which X0 through X3 contain data and X4 is the parity disk. Suppose that a write is performed that only involves a strip on disk X1. Initially, for each bit *i*, we have the following relationship:

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \quad \textbf{(6.2)}$$

After the update, with potentially altered bits indicated by a prime symbol:

$$\begin{aligned} X4'(i) &= X3(i) \oplus X2(i) \oplus X1'(i)X0(i) \\ &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \oplus X1(i) \oplus X1(i) \\ &= X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \oplus X1(i) \oplus X1'(i) \\ &= X4(i) \oplus X1(i) \oplus X1'(i) \end{aligned}$$

The preceding set of equations is derived as follows. The first line shows that a change in X1 will also affect the parity disk X4. In the second line, we add the terms $\oplus$ X1(i) $\oplus$ X1(i)]. Because the exclusive-OR of any quantity with itself is 0, this does not affect the equation. However, it is a convenience that is used to create the third line, by reordering. Finally, Equation (6.2) is used to replace the first four terms by X4(i).

To calculate the new parity, the array management software must read the old user strip and the old parity strip. Then it can update these two strips with the new data and the newly calculated parity. Thus, each strip write involves two reads and two writes.

In the case of a larger size I/O write that involves strips on all disk drives, parity is easily computed by calculation using only the new data bits. Thus, the parity drive can be updated in parallel with the data drives and there are no extra reads or writes.

In any case, every write operation must involve the parity disk, which therefore can become a bottleneck.

RAID Level 5

RAID 5 is organized in a similar fashion to RAID 4. The difference is that RAID 5 distributes the parity strips across all disks. A typical allocation is a round-robin scheme, as illustrated in Figure 6.6f. For an n-disk array, the parity strip is on a different disk for the first n stripes, and the pattern then repeats.

The distribution of parity strips across all drives avoids the potential I/O bottle-neck found in RAID 4.

RAID Level 6

RAID 6 was introduced in a subsequent paper by the Berkeley researchers [KATZ89]. In the RAID 6 scheme, two different parity calculations are carried out and stored in separate blocks on different disks. Thus, a RAID 6 array whose user data require N disks consists of $N + 2$ disks.

Figure 6.6g illustrates the scheme. P and Q are two different data check algorithms. One of the two is the exclusive-OR calculation used in RAID 4 and 5. But the other is an independent data check algorithm. This makes it possible to regenerate data even if two disks containing user data fail.

The advantage of RAID 6 is that it provides extremely high data availability. Three disks would have to fail within the MTTR (mean time to repair) interval to cause data to be lost. On the other hand, RAID 6 incurs a substantial write penalty, because each write affects two parity blocks. Performance benchmarks [EISC07] show a RAID 6 controller can suffer more than a 30% drop in overall write performance compared with a RAID 5 implementation. RAID 5 and RAID 6 read performance is comparable.

Table 6.4 is a comparative summary of the seven levels.

6.3 SOLID STATE DRIVES

One of the most significant developments in computer architecture in recent years is the increasing use of solid state drives (SSDs) to complement or even replace **hard disk drives (HDDs)**, both as internal and external secondary memory. The term *solid*

Table 6.4 RAID Comparison

Level	Advantages	Disadvantages	Applications
0	I/O performance is greatly improved by spreading the I/O load across many channels and drives No parity calculation overhead is involved Very simple design Easy to implement	The failure of just one drive will result in all data in an array being lost	Video production and editing Image Editing Pre-press applications Any application requiring high bandwidth
1	100% redundancy of data means no rebuild is necessary in case of a disk failure, just a copy to the replacement disk Under certain circumstances, RAID 1 can sustain multiple simultaneous drive failures Simplest RAID storage subsystem design	Highest disk overhead of all RAID types (100%)—inefficient	Accounting Payroll Financial Any application requiring very high availability
2	Extremely high data transfer rates possible The higher the data transfer rate required, the better the ratio of data disks to ECC disks Relatively simple controller design compared to RAID levels 3, 4, & 5	Very high ratio of ECC disks to data disks with smaller word sizes—inefficient Entry level cost very high—requires very high transfer rate requirement to justify	No commercial implementations exist/not commercially viable
3	Very high read data transfer rate Very high write data transfer rate Disk failure has an insignificant impact on throughput Low ratio of ECC (parity) disks to data disks means high efficiency	Transaction rate equal to that of a single disk drive at best (if spindles are synchronized) Controller design is fairly complex	Video production and live streaming Image editing Video editing Prepress applications Any application requiring high throughput
4	Very high Read data transaction rate Low ratio of ECC (parity) disks to data disks means high efficiency	Quite complex controller design Worst write transaction rate and Write aggregate transfer rate Difficult and inefficient data rebuild in the event of disk failure	No commercial implementations exist/not commercially viable
5	Highest Read data transaction rate Low ratio of ECC (parity) disks to data disks means high efficiency Good aggregate transfer rate	Most complex controller design Difficult to rebuild in the event of a disk failure (as compared to RAID level 1)	File and application servers Database servers Web, e-mail, and news servers Intranet servers Most versatile RAID level
6	Provides for an extremely high data fault tolerance and can sustain multiple simultaneous drive failures	More complex controller design Controller overhead to compute parity addresses is extremely high	Perfect solution for mission critical applications

state refers to electronic circuitry built with semiconductors. An SSD is a memory device made with solid state components that can be used as a replacement to a hard disk drive. The SSDs now on the market and coming on line use NAND flash memory, which is described in Chapter 5.

SSD Compared to HDD

As the cost of flash-based SSDs has dropped and the performance and bit density increased, SSDs have become increasingly competitive with HDDs. Table 6.5 shows typical measures of comparison at the time of this writing.

SSDs have the following advantages over HDDs:

- **High-performance input/output operations per second (IOPS):** Significantly increases performance I/O subsystems.
- **Durability:** Less susceptible to physical shock and vibration.
- **Longer lifespan:** SSDs are not susceptible to mechanical wear.
- **Lower power consumption:** SSDs use considerably less power than comparable-size HDDs.
- **Quieter and cooler running capabilities:** Less space required, lower energy costs, and a greener enterprise.
- **Lower access times and latency rates:** Over 10 times faster than the spinning disks in an HDD.

Currently, HDDs enjoy a cost per bit advantage and a capacity advantage, but these differences are shrinking.

SSD Organization

Figure 6.8 illustrates a general view of the common architectural system component associated with any SSD system. On the host system, the operating system invokes file system software to access data on the disk. The file system, in turn, invokes I/O driver software. The I/O driver software provides host access to the particular SSD product. The interface component in Figure 6.8 refers to the physical and electrical interface between the host processor and the SSD peripheral device. If the device is an internal hard drive, a common interface is PCIe. For external devices, one common interface is USB.

Table 6.5 Comparison of Solid State Drives and Disk Drives

	NAND Flash Drives	Seagate Laptop Internal HDD
File copy/write speed	200–550 Mbps	50–120 Mbps
Power draw/battery life	Less power draw, averages 2–3 watts, resulting in 30+ minute battery boost	More power draw, averages 6–7 watts and therefore uses more battery
Storage capacity	Typically not larger than 512 GB for notebook size drives; 1 TB max for desktops	Typically around 500 GB and 2 TB max for notebook size drives; 4 TB max for desktops
Cost	Approx. $0.50 per GB for a 1-TB drive	Approx. $0.15 per GB for a 4-TB drive

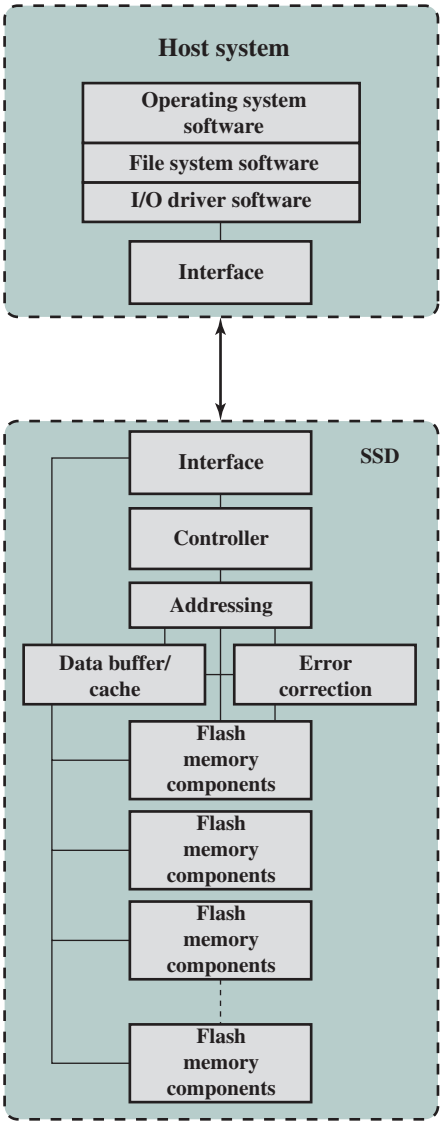


Figure 6.8 Solid State Drive Architecture

In addition to the interface to the host system, the SSD contains the following components:

- **Controller:** Provides SSD device level interfacing and firmware execution.
- **Addressing:** Logic that performs the selection function across the flash memory components.
- **Data buffer/cache:** High speed RAM memory components used for speed matching and to increased data throughput.

- **Error correction:** Logic for error detection and correction.
- **Flash memory components:** Individual NAND flash chips.

Practical Issues

There are two practical issues peculiar to SSDs that are not faced by HDDs. First, SSD performance has a tendency to slow down as the device is used. To understand the reason for this, you need to know that files are stored on disk as a set of pages, typically 4 KB in length. These pages are not necessarily, and indeed not typically, stored as a contiguous set of pages on the disk. The reason for this arrangement is explained in our discussion of virtual memory in Chapter 8. However, flash memory is accessed in blocks, with a typical block size of 512 KB, so that there are typically 128 pages per block. Now consider what must be done to write a page onto a flash memory.

1. The entire block must be read from the flash memory and placed in a RAM buffer. Then the appropriate page in the RAM buffer is updated.
2. Before the block can be written back to flash memory, the entire block of flash memory must be erased—it is not possible to erase just one page of the flash memory.
3. The entire block from the buffer is now written back to the flash memory.

Now, when a flash drive is relatively empty and a new file is created, the pages of that file are written on to the drive contiguously, so that one or only a few blocks are affected. However, over time, because of the way virtual memory works, files become fragmented, with pages scattered over multiple blocks. As the drive become more occupied, there is more fragmentation, so the writing of a new file can affect multiple blocks. Thus, the writing of multiple pages from one block becomes slower, the more fully occupied the disk is. Manufacturers have developed a variety of techniques to compensate for this property of flash memory, such as setting aside a substantial portion of the SSD as extra space for write operations (called overprovisioning), then to erase inactive pages during idle time used to defragment the disk. Another technique is the TRIM command, which allows an operating system to inform an SSD which blocks of data are no longer considered in use and can be wiped internally.[4]

A second practical issue with flash memory drives is that a flash memory becomes unusable after a certain number of writes. As flash cells are stressed, they lose their ability to record and retain values. A typical limit is 100,000 writes [GSOE08]. Techniques for prolonging the life of an SSD drive include front-ending the flash with a cache to delay and group write operations, using wear-leveling algorithms that evenly distribute writes across block of cells, and sophisticated bad-block management techniques. In addition, vendors are deploying SSDs in RAID configurations to further reduce the probability of data loss. Most flash devices are also capable of estimating their own remaining lifetimes so systems can anticipate failure and take preemptive action.

[4]While TRIM is frequently spelled in capital letters, it is not an acronym; it is merely a command name.

6.4 OPTICAL MEMORY

In 1983, one of the most successful consumer products of all time was introduced: the compact disk (CD) digital audio system. The CD is a nonerasable disk that can store more than 60 minutes of audio information on one side. The huge commercial success of the CD enabled the development of low-cost optical-disk storage technology that has revolutionized computer data storage. A variety of optical-disk systems have been introduced (Table 6.6). We briefly review each of these.

Compact Disk

CD-ROM Both the audio CD and the **CD-ROM** (compact disk read-only memory) share a similar technology. The main difference is that CD-ROM players are more rugged and have error correction devices to ensure that data are properly transferred from disk to computer. Both types of disk are made the same way. The disk is formed from a resin, such as polycarbonate. Digitally recorded information (either music or computer data) is imprinted as a series of microscopic pits on the surface of the polycarbonate. This is done, first of all, with a finely focused, high-intensity laser to create a master disk. The master is used, in turn, to make a die to stamp out copies onto polycarbonate. The pitted surface is then coated with a highly reflective surface, usually aluminum or gold. This shiny surface is protected against dust and scratches by a top coat of clear acrylic. Finally, a label can be silkscreened onto the acrylic.

Table 6.6 Optical Disk Products

CD	Compact Disk. A nonerasable disk that stores digitized audio information. The standard system uses 12-cm disks and can record more than 60 minutes of uninterrupted playing time.
CD-ROM	Compact Disk Read-Only Memory. A nonerasable disk used for storing computer data. The standard system uses 12-cm disks and can hold more than 650 Mbytes.
CD-R	CD Recordable. Similar to a CD-ROM. The user can write to the disk only once.
CD-RW	CD Rewritable. Similar to a CD-ROM. The user can erase and rewrite to the disk multiple times.
DVD	Digital Versatile Disk. A technology for producing digitized, compressed representation of video information, as well as large volumes of other digital data. Both 8 and 12 cm diameters are used, with a double-sided capacity of up to 17 Gbytes. The basic DVD is read-only (DVD-ROM).
DVD-R	DVD Recordable. Similar to a DVD-ROM. The user can write to the disk only once. Only one-sided disks can be used.
DVD-RW	DVD Rewritable. Similar to a DVD-ROM. The user can erase and rewrite to the disk multiple times. Only one-sided disks can be used.
Blu-ray DVD	High-definition video disk. Provides considerably greater data storage density than DVD, using a 405-nm (blue-violet) laser. A single layer on a single side can store 25 Gbytes.

Information is retrieved from a CD or CD-ROM by a low-powered laser housed in an optical-disk player, or drive unit. The laser shines through the clear polycarbonate while a motor spins the disk past it (Figure 6.9). The intensity of the reflected light of the laser changes as it encounters a **pit**. Specifically, if the laser beam falls on a pit, which has a somewhat rough surface, the light scatters and a low intensity is reflected back to the source. The areas between pits are called **lands**. A land is a smooth surface, which reflects back at higher intensity. The change between pits and lands is detected by a photosensor and converted into a digital signal. The sensor tests the surface at regular intervals. The beginning or end of a pit represents a 1; when no change in elevation occurs between intervals, a 0 is recorded.

Recall that on a magnetic disk, information is recorded in concentric tracks. With the simplest constant angular velocity (CAV) system, the number of bits per track is constant. An increase in density is achieved with **multiple zone recording**, in which the surface is divided into a number of zones, with zones farther from the center containing more bits than zones closer to the center. Although this technique increases capacity, it is still not optimal.

To achieve greater capacity, CDs and CD-ROMs do not organize information on concentric tracks. Instead, the disk contains a single spiral track, beginning near the center and spiraling out to the outer edge of the disk. Sectors near the outside of the disk are the same length as those near the inside. Thus, information is packed evenly across the disk in segments of the same size and these are scanned at the same rate by rotating the disk at a variable speed. The pits are then read by the laser at a **constant linear velocity (CLV)**. The disk rotates more slowly for accesses near the outer edge than for those near the center. Thus, the capacity of a track and the rotational delay both increase for positions nearer the outer edge of the disk. The data capacity for a CD-ROM is about 680 MB.

Data on the CD-ROM are organized as a sequence of blocks. A typical block format is shown in Figure 6.10. It consists of the following fields:

- **Sync:** The sync field identifies the beginning of a block. It consists of a byte of all 0s, 10 bytes of all 1s, and a byte of all 0s.
- **Header:** The header contains the block address and the mode byte. Mode 0 specifies a blank data field; mode 1 specifies the use of an error-correcting

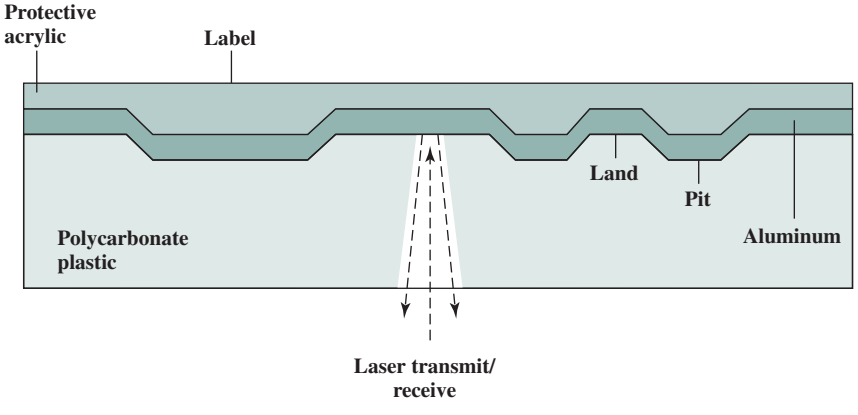


Figure 6.9 CD Operation

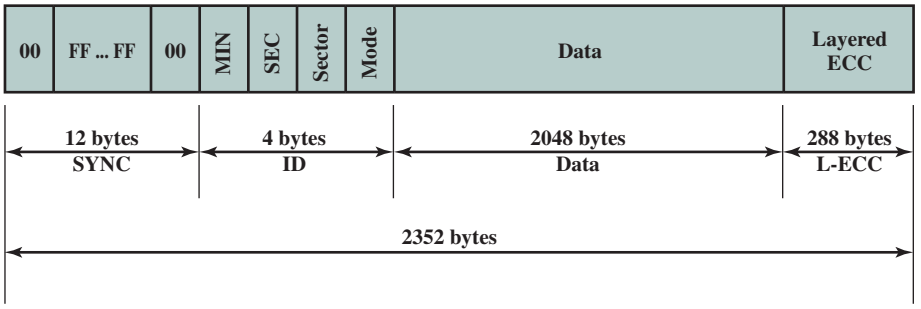


Figure 6.10 CD-ROM Block Format

code and 2048 bytes of data; mode 2 specifies 2336 bytes of user data with no error-correcting code.

- **Data:** User data.
- **Auxiliary:** Additional user data in mode 2. In mode 1, this is a 288-byte error-correcting code.

With the use of CLV, random access becomes more difficult. Locating a specific address involves moving the head to the general area, adjusting the rotation speed and reading the address, and then making minor adjustments to find and access the specific sector.

CD-ROM is appropriate for the distribution of large amounts of data to a large number of users. Because of the expense of the initial writing process, it is not appropriate for individualized applications. Compared with traditional magnetic disks, the CD-ROM has two advantages:

- The optical disk together with the information stored on it can be mass replicated inexpensively—unlike a magnetic disk. The database on a magnetic disk has to be reproduced by copying one disk at a time using two disk drives.
- The optical disk is removable, allowing the disk itself to be used for archival storage. Most magnetic disks are nonremovable. The information on nonremovable magnetic disks must first be copied to another storage medium before the disk drive/disk can be used to store new information.

The disadvantages of CD-ROM are as follows:

- It is read-only and cannot be updated.
- It has an access time much longer than that of a magnetic disk drive, as much as half a second.

CD RECORDABLE To accommodate applications in which only one or a small number of copies of a set of data is needed, the write-once read-many CD, known as the **CD recordable (CD-R),** has been developed. For CD-R, a disk is prepared in such a way that it can be subsequently written once with a laser beam of modest-intensity. Thus, with a somewhat more expensive disk controller than for CD-ROM, the customer can write once as well as read the disk.

The CD-R medium is similar to but not identical to that of a CD or CD-ROM. For CDs and CD-ROMs, information is recorded by the pitting of the surface

of the medium, which changes reflectivity. For a CD-R, the medium includes a dye layer. The dye is used to change reflectivity and is activated by a high-intensity laser. The resulting disk can be read on a CD-R drive or a CD-ROM drive.

The CD-R optical disk is attractive for archival storage of documents and files. It provides a permanent record of large volumes of user data.

CD REWRITABLE The **CD-RW** optical disk can be repeatedly written and overwritten, as with a magnetic disk. Although a number of approaches have been tried, the only pure optical approach that has proved attractive is called **phase change**. The phase change disk uses a material that has two significantly different reflectivities in two different phase states. There is an amorphous state, in which the molecules exhibit a random orientation that reflects light poorly; and a crystalline state, which has a smooth surface that reflects light well. A beam of laser light can change the material from one phase to the other. The primary disadvantage of phase change optical disks is that the material eventually and permanently loses its desirable properties. Current materials can be used for between 500,000 and 1,000,000 erase cycles.

The CD-RW has the obvious advantage over CD-ROM and CD-R that it can be rewritten and thus used as a true secondary storage. As such, it competes with magnetic disk. A key advantage of the optical disk is that the engineering tolerances for optical disks are much less severe than for high-capacity magnetic disks. Thus, they exhibit higher reliability and longer life.

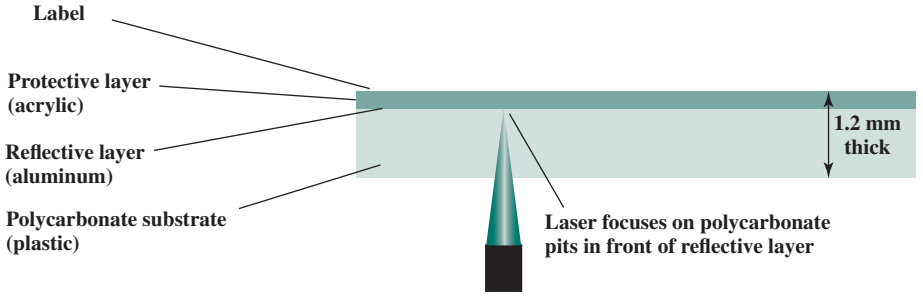
Digital Versatile Disk

With the capacious **digital versatile disk (DVD),** the electronics industry has at last found an acceptable replacement for the analog VHS video tape. The DVD has replaced the videotape used in video cassette recorders (VCRs) and, more important for this discussion, replaced the CD-ROM in personal computers and servers. The DVD takes video into the digital age. It delivers movies with impressive picture quality, and it can be randomly accessed like audio CDs, which DVD machines can also play. Vast volumes of data can be crammed onto the disk, currently seven times as much as a CD-ROM. With DVD's huge storage capacity and vivid quality, PC games have become more realistic and educational software incorporates more video. Following in the wake of these developments has been a new crest of traffic over the Internet and corporate intranets, as this material is incorporated into Web sites.

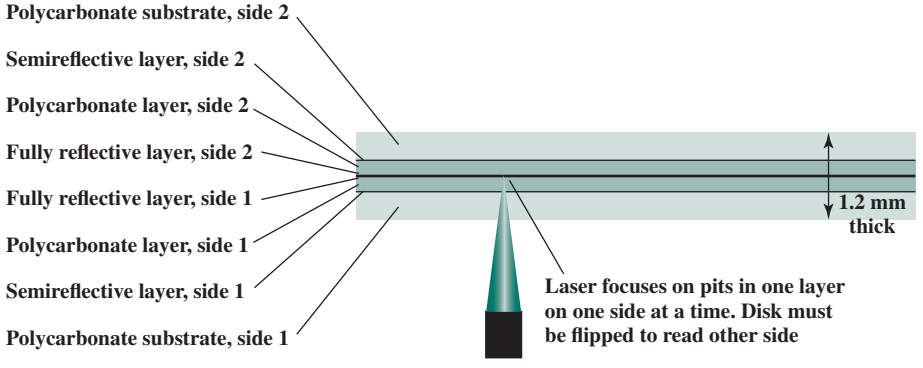
The DVD's greater capacity is due to three differences from CDs (Figure 6.11):

1. Bits are packed more closely on a DVD. The spacing between loops of a spiral on a CD is 1.6 μm and the minimum distance between pits along the spiral is 0.834 μm.

   The DVD uses a laser with shorter wavelength and achieves a loop spacing of 0.74 μm and a minimum distance between pits of 0.4 μm. The result of these two improvements is about a seven-fold increase in capacity, to about 4.7 GB.
2. The DVD employs a second layer of pits and lands on top of the first layer. A dual-layer DVD has a semireflective layer on top of the reflective layer, and by adjusting focus, the lasers in DVD drives can read each layer separately. This technique almost doubles the capacity of the disk, to about 8.5 GB. The lower reflectivity of the second layer limits its storage capacity so that a full doubling is not achieved.



(a) CD-ROM–Capacity 682 MB


Polycarbonate substrate, side 2
Semireflective layer, side 2
Polycarbonate layer, side 2
Fully reflective layer, side 2
Fully reflective layer, side 1
Polycarbonate layer, side 1
Semireflective layer, side 1
Polycarbonate substrate, side 1
1.2 mm thick
Laser focuses on pits in one layer on one side at a time. Disk must be flipped to read other side

(b) DVD-ROM, double-sided, dual-layer–Capacity 17 GB

Figure 6.11 CD-ROM and DVD-ROM

3. The **DVD-ROM** can be two sided, whereas data are recorded on only one side of a CD. This brings total capacity up to 17 GB.

As with the CD, DVDs come in writeable as well as read-only versions (Table 6.6).

High-Definition Optical Disks

High-definition optical disks are designed to store high-definition videos and to provide significantly greater storage capacity compared to DVDs. The higher bit density is achieved by using a laser with a shorter wavelength, in the blue-violet range. The data pits, which constitute the digital 1s and 0s, are smaller on the high-definition optical disks compared to DVD because of the shorter laser wavelength.

Two competing disk formats and technologies initially competed for market acceptance: HD DVD and **Blu-ray** DVD. The Blu-ray scheme ultimately achieved market dominance. The HD DVD scheme can store 15 GB on a single layer on a single side. Blu-ray positions the data layer on the disk closer to the laser (shown on the right-hand side of each diagram in Figure 6.12). This enables a tighter focus and less distortion and thus smaller pits and tracks. Blu-ray can store 25 GB on a single layer. Three versions are available: read only (BD-ROM), recordable once (BD-R), and rerecordable (BD-RE).

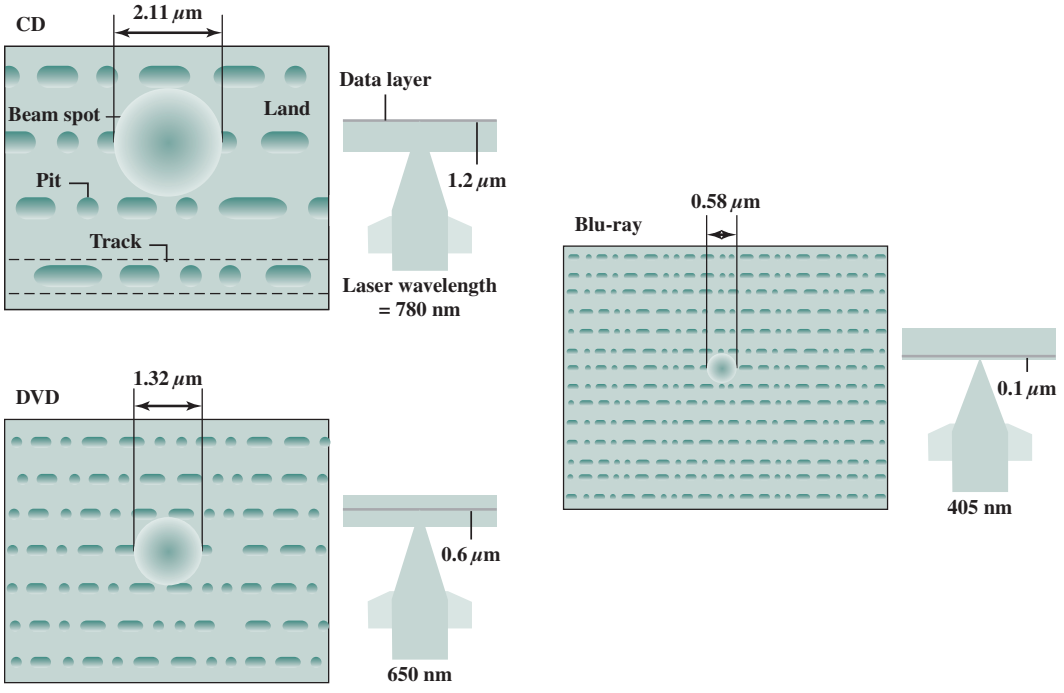


Figure 6.12 Optical Memory Characteristics

6.5 MAGNETIC TAPE

Tape systems use the same reading and recording techniques as disk systems. The medium is flexible polyester (similar to that used in some clothing) tape coated with magnetizable material. The coating may consist of particles of pure metal in special binders or vapor-plated metal films. The tape and the tape drive are analogous to a home tape recorder system. Tape widths vary from 0.38 cm (0.15 inch) to 1.27 cm (0.5 inch). Tapes used to be packaged as open reels that have to be threaded through a second spindle for use. Today, virtually all tapes are housed in cartridges.

Data on the tape are structured as a number of parallel tracks running lengthwise. Earlier tape systems typically used nine tracks. This made it possible to store data one byte at a time, with an additional parity bit as the ninth track. This was followed by tape systems using 18 or 36 tracks, corresponding to a digital word or double word. The recording of data in this form is referred to as **parallel recording**. Most modern systems instead use **serial recording**, in which data are laid out as a sequence of bits along each track, as is done with magnetic disks. As with the disk, data are read and written in contiguous blocks, called *physical records,* on a tape. Blocks on the tape are separated by gaps referred to as *interrecord* gaps. As with the disk, the tape is formatted to assist in locating physical records.

The typical recording technique used in serial tapes is referred to as **serpentine recording.** In this technique, when data are being recorded, the first set of bits is recorded along the whole length of the tape. When the end of the tape is reached,

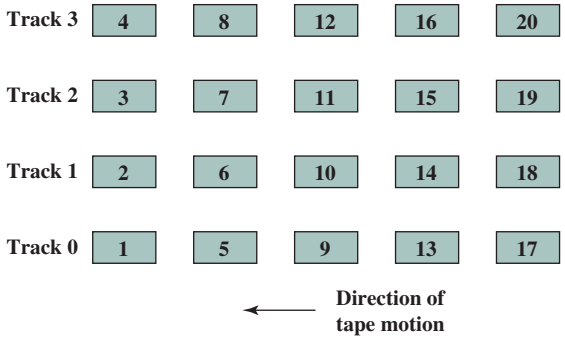
the heads are repositioned to record a new track, and the tape is again recorded on its whole length, this time in the opposite direction. That process continues, back and forth, until the tape is full (Figure 6.13a). To increase speed, the read-write head is capable of reading and writing a number of adjacent tracks simultaneously (typically two to eight tracks). Data are still recorded serially along individual tracks, but blocks in sequence are stored on adjacent tracks, as suggested by Figure 6.13b.

A tape drive is a *sequential-access* device. If the tape head is positioned at record 1, then to read record *N,* it is necessary to read physical records 1 through $N-1$, one at a time. If the head is currently positioned beyond the desired record, it is necessary to rewind the tape a certain distance and begin reading forward. Unlike the disk, the tape is in motion only during a read or write operation.

In contrast to the tape, the disk drive is referred to as a *direct-access* device. A disk drive need not read all the sectors on a disk sequentially to get to the desired one. It must only wait for the intervening sectors within one track and can make successive accesses to any track.

Magnetic tape was the first kind of secondary memory. It is still widely used as the lowest-cost, slowest-speed member of the memory hierarchy.

(a) Serpentine reading and writing



(b) Block layout for system that reads–writes four tracks simultaneously

Figure 6.13 Typical Magnetic Tape Features

Table 6.7 LTO Tape Drives

	LTO-1	LTO-2	LTO-3	LTO-4	LTO-5	LTO-6	LTO-7	LTO-8
Release date	2000	2003	2005	2007	2010	2012	TBA	TBA
Compressed capacity	200 GB	400 GB	800 GB	1600 GB	3.2 TB	8 TB	16 TB	32 TB
Compressed transfer rate	40 MB/s	80 MB/s	160 MB/s	240 MB/s	280 MB/s	400 MB/s	788 MB/s	1.18 GB/s
Linear density (bits/mm)	4880	7398	9638	13,250	15,142	15,143		
Tape tracks	384	512	704	896	1280	2176		
Tape length (m)	609	609	680	820	846	846		
Tape width (cm)	1.27	1.27	1.27	1.27	1.27	1.27		
Write elements	8	8	16	16	16	16		
WORM?	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Encryption Capable?	No	No	No	Yes	Yes	Yes	Yes	Yes
Partitioning?	No	No	No	No	Yes	Yes	Yes	Yes

The dominant tape technology today is a cartridge system known as linear tape-open (LTO). LTO was developed in the late 1990s as an open-source alternative to the various proprietary systems on the market. Table 6.7 shows parameters for the various LTO generations. See Appendix J for details.

6.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

access time	DVD-RW	optical memory
Blu-ray	fixed-head disk	pit
CD	flash memory	platter
CD-R	floppy disk	RAID
CD-ROM	gap	removable disk
CD-RW	hard disk drive (HDD)	rotational delay
constant angular velocity (CAV)	head	sector
	land	seek time
constant linear velocity (CLV)	magnetic disk	serpentine recording
	magnetic tape	solid state drive (SSD)
cylinder	magnetoresistive	striped data
DVD	movable-head disk	substrate
DVD-R	multiple zone recording	track
DVD-ROM	nonremovable disk	transfer time

Review Questions

6.1 What are the advantages of using a glass substrate for a magnetic disk?
6.2 How are data written onto a magnetic disk?
6.3 How are data read from a magnetic disk?
6.4 Explain the difference between a simple CAV system and a multiple zone recording system.
6.5 Define the terms *track, cylinder,* and *sector*.
6.6 What is the typical disk sector size?
6.7 Define the terms *seek time, rotational delay, access time,* and *transfer time*.
6.8 What common characteristics are shared by all RAID levels?
6.9 Briefly define the seven RAID levels.
6.10 Explain the term *striped data*.
6.11 How is redundancy achieved in a RAID system?
6.12 In the context of RAID, what is the distinction between parallel access and independent access?
6.13 What is the difference between CAV and CLV?
6.14 What differences between a CD and a DVD account for the larger capacity of the latter?
6.15 Explain serpentine recording.

Problems

6.1 Justify Equation 6.1. That is, explain how each of the three terms on the right-hand side of the equation contributes to the value on the left-hand side.

6.2 Consider a disk with N tracks numbered from 0 to $(N - 1)$ and assume that requested sectors are distributed randomly and evenly over the disk. We want to calculate the average number of tracks traversed by a seek.

a. First, calculate the probability of a seek of length j when the head is currently positioned over track t. *Hint*: This is a matter of determining the total number of combinations, recognizing that all track positions for the destination of the seek are equally likely.

b. Next, calculate the probability of a seek of length K. *Hint*: This involves the summing over all possible combinations of movements of K tracks.

c. Calculate the average number of tracks traversed by a seek, using the formula for expected value

$$\mathrm{E}[x] = \sum_{i=0}^{N-1} i \times \Pr[x = i]$$

Hint: Use the equalities: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$; $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.

d. Show that for large values of N, the average number of tracks traversed by a seek approaches $N/3$.

6.3 Define the following for a disk system:

t_s = seek time; average time to position head over track
r = rotation speed of the disk, in revolutions per second
n = number of bits per sector
N = capacity of a track, in bits
t_{sector} = time to access a sector

Develop a formula for t_{sector} as a function of the other parameters.

6.4 Consider a magnetic disk drive with 8 surfaces, 512 tracks per surface, and 64 sectors per track. Sector size is 1 kB. The average seek time is 8 ms, the track-to-track access time is 1.5 ms, and the drive rotates at 3600 rpm. Successive tracks in a cylinder can be read without head movement.
- **a.** What is the disk capacity?
- **b.** What is the average access time? Assume this file is stored in successive sectors and tracks of successive cylinders, starting at sector 0, track 0, of cylinder *i*.
- **c.** Estimate the time required to transfer a 5-MB file.
- **d.** What is the burst transfer rate?

6.5 Consider a single-platter disk with the following parameters: rotation speed: 7200 rpm; number of tracks on one side of platter: 30,000; number of sectors per track: 600; seek time: one ms for every hundred tracks traversed. Let the disk receive a request to access a random sector on a random track and assume the disk head starts at track 0.
- **a.** What is the average seek time?
- **b.** What is the average rotational latency?
- **c.** What is the transfer time for a sector?
- **d.** What is the total average time to satisfy a request?

6.6 A distinction is made between physical records and logical records. A **logical record** is a collection of related data elements treated as a conceptual unit, independent of how or where the information is stored. A **physical record** is a contiguous area of storage space that is defined by the characteristics of the storage device and operating system. Assume a disk system in which each physical record contains thirty 120-byte logical records. Calculate how much disk space (in sectors, tracks, and surfaces) will be required to store 300,000 logical records if the disk is fixed-sector with 512 bytes/sector, with 96 sectors/track, 110 tracks per surface, and 8 usable surfaces. Ignore any file header record(s) and track indexes, and assume that records cannot span two sectors.

6.7 Consider a disk that rotates at 3600 rpm. The seek time to move the head between adjacent tracks is 2 ms. There are 32 sectors per track, which are stored in linear order from sector 0 through sector 31. The head sees the sectors in ascending order. Assume the read/write head is positioned at the start of sector 1 on track 8. There is a main memory buffer large enough to hold an entire track. Data is transferred between disk locations by reading from the source track into the main memory buffer and then writing the data from the buffer to the target track.
- **a.** How long will it take to transfer sector 1 on track 8 to sector 1 on track 9?
- **b.** How long will it take to transfer all the sectors of track 8 to the corresponding sectors of track 9?

6.8 It should be clear that disk striping can improve data transfer rate when the strip size is small compared to the I/O request size. It should also be clear that RAID 0 provides improved performance relative to a single large disk, because multiple I/O requests can be handled in parallel. However, in this latter case, is disk striping necessary? That is, does disk striping improve I/O request rate performance compared to a comparable disk array without striping?

6.9 Consider a 4-drive, 200 GB-per-drive RAID array. What is the available data storage capacity for each of the RAID levels 0, 1, 3, 4, 5, and 6?

6.10 For a compact disk, audio is converted to digital with 16-bit samples, and is treated a stream of 8-bit bytes for storage. One simple scheme for storing this data, called direct recording, would be to represent a 1 by a land and a 0 by a pit. Instead, each byte is expanded into a 14-bit binary number. It turns out that exactly 256 (2^8) of the total of 16,134 (2^{14}) 14-bit numbers have at least two 0s between every pair of 1s, and these are the numbers selected for the expansion from 8 to 14 bits. The optical system detects the presence of 1s by detecting a transition for pit to land or land to pit. It detects 0s by measuring the distances between intensity changes. This scheme requires that there are no 1s in succession; hence the use of the 8-to-14 code.

The advantage of this scheme is as follows. For a given laser beam diameter, there is a minimum-pit size, regardless of how the bits are represented. With this scheme, this minimum-pit size stores 3 bits, because at least two 0s follow every 1. With direct recording, the same pit would be able to store only one bit. Considering both the number of bits stored per pit and the 8-to-14 bit expansion, which scheme stores the most bits and by what factor?

6.11 Design a backup strategy for a computer system. One option is to use plug-in external disks, which cost $150 for each 500 GB drive. Another option is to buy a tape drive for $2500, and 400 GB tapes for $50 apiece. (These were realistic prices in 2008.) A typical backup strategy is to have two sets of backup media onsite, with backups alternately written on them so in case the system fails while making a backup, the previous version is still intact. There's also a third set kept offsite, with the offsite set periodically swapped with an on-site set.

a. Assume you have 1 TB (1000 GB) of data to back up. How much would a disk backup system cost?

b. How much would a tape backup system cost for 1 TB?

c. How large would each backup have to be in order for a tape strategy to be less expensive?

d. What kind of backup strategy favors tapes?

CHAPTER 7

INPUT/OUTPUT

7.1 External Devices

7.2 I/O Modules

7.3 Programmed I/O

7.4 Interrupt-Driven I/O

7.5 Direct Memory Access

7.6 Direct Cache Access

7.7 I/O Channels and Processors

7.8 External Interconnection Standards

7.9 IBM zEnterprise EC12 I/O Structure

7.10 Key Terms, Review Questions, and Problems

Learning Objectives

After studying this chapter, you should be able to:

- Explain the use of I/O modules as part of a computer organization.
- Understand the difference between **programmed I/O** and **interrupt-driven I/O** and discuss their relative merits.
- Present an overview of the operation of direct memory access.
- Present an overview of direct cache access.
- Explain the function and use of I/O channels.

I/O System Design Tool

In addition to the processor and a set of memory modules, the third key element of a computer system is a set of I/O modules. Each module interfaces to the system bus or central switch and controls one or more peripheral devices. An I/O module is not simply a set of mechanical connectors that wire a device into the system bus. Rather, the I/O module contains logic for performing a communication function between the peripheral and the bus.

The reader may wonder why one does not connect peripherals directly to the system bus. The reasons are as follows:

- There are a wide variety of peripherals with various methods of operation. It would be impractical to incorporate the necessary logic within the processor to control a range of devices.
- The data transfer rate of peripherals is often much slower than that of the memory or processor. Thus, it is impractical to use the high-speed system bus to communicate directly with a peripheral.
- On the other hand, the data transfer rate of some peripherals is faster than that of the memory or processor. Again, the mismatch would lead to inefficiencies if not managed properly.
- Peripherals often use different data formats and word lengths than the computer to which they are attached.
- Thus, an I/O module is required. This module has two major functions (Figure 7.1):
  - Interface to the processor and memory via the system bus or central switch.
  - Interface to one or more peripheral devices by tailored data links.

We begin this chapter with a brief discussion of external devices, followed by an overview of the structure and function of an I/O module. Then we look at the various ways in which the I/O function can be performed in cooperation with the processor and memory: the internal I/O interface. Next, we examine in some

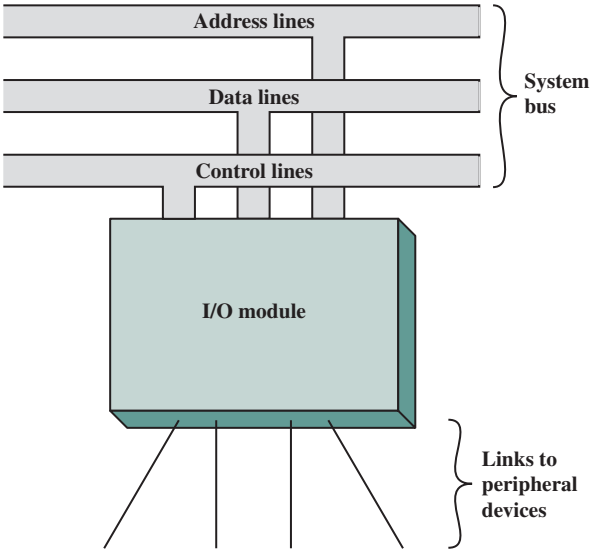


Figure 7.1 Generic Model of an I/O Module

detail direct memory access and the more recent innovation of direct cache access. Finally, we examine the external I/O interface, between the I/O module and the outside world.

7.1 EXTERNAL DEVICES

I/O operations are accomplished through a wide assortment of external devices that provide a means of exchanging data between the external environment and the computer. An external device attaches to the computer by a link to an I/O module (Figure 7.1). The link is used to exchange control, status, and data between the I/O module and the external device. An external device connected to an I/O module is often referred to as a *peripheral device* or, simply, a *peripheral.*

We can broadly classify external devices into three categories:

- **Human readable:** Suitable for communicating with the computer user;
- **Machine readable:** Suitable for communicating with equipment;
- **Communication:** Suitable for communicating with remote devices.

Examples of human-readable devices are video display terminals (VDTs) and printers. Examples of machine-readable devices are magnetic disk and tape systems, and sensors and actuators, such as are used in a robotics application. Note that we are viewing disk and tape systems as I/O devices in this chapter, whereas in Chapter 6 we viewed them as memory devices. From a functional point of view, these devices are part of the memory hierarchy, and their use is appropriately discussed in Chapter 6. From a structural point of view, these devices are controlled by I/O modules and are hence to be considered in this chapter.

Communication devices allow a computer to exchange data with a remote device, which may be a human-readable device, such as a terminal, a machine-readable device, or even another computer.

In very general terms, the nature of an external device is indicated in Figure 7.2. The interface to the I/O module is in the form of control, data, and status signals. *Control signals* determine the function that the device will perform, such as send data to the I/O module (INPUT or READ), accept data from the I/O module (OUTPUT or WRITE), report status, or perform some control function particular to the device (e.g., position a disk head). *Data* are in the form of a set of bits to be sent to or received from the I/O module. *Status signals* indicate the state of the device. Examples are READY/NOT-READY to show whether the device is ready for data transfer.

Control logic associated with the device controls the device's operation in response to direction from the I/O module. The *transducer* converts data from electrical to other forms of energy during output and from other forms to electrical during input. Typically, a buffer is associated with the transducer to temporarily hold data being transferred between the I/O module and the external environment. A buffer size of 8 to 16 bits is common for serial devices, whereas block-oriented devices such as disk drive controllers may have much larger buffers.

The interface between the I/O module and the external device will be examined in Section 7.7. The interface between the external device and the environment is beyond the scope of this book, but several brief examples are given here.

Keyboard/Monitor

The most common means of computer/user interaction is a keyboard/monitor arrangement. The user provides input through the keyboard, the input is then transmitted to the computer and may also be displayed on the monitor. In addition, the monitor displays data provided by the computer.

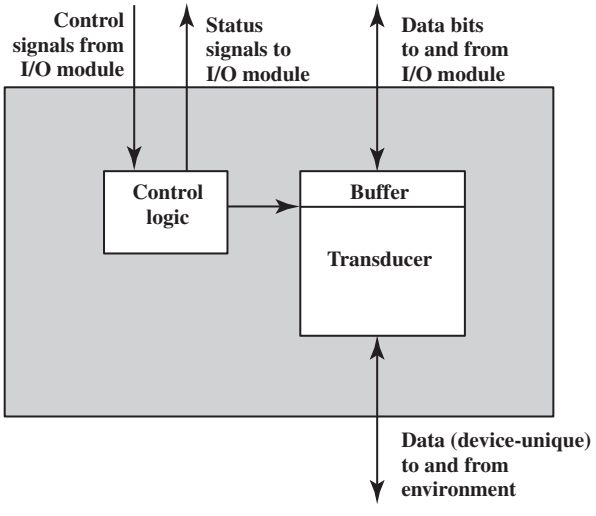


Figure 7.2 Block Diagram of an External Device

The basic unit of exchange is the character. Associated with each character is a code, typically 7 or 8 bits in length. The most commonly used text code is the International Reference Alphabet (IRA).[1] Each character in this code is represented by a unique 7-bit binary code; thus, 128 different characters can be represented. Characters are of two types: printable and control. Printable characters are the alphabetic, numeric, and special characters that can be printed on paper or displayed on a screen. Some of the control characters have to do with controlling the printing or displaying of characters; an example is carriage return. Other control characters are concerned with communications procedures. See Appendix H for details.

For keyboard input, when the user depresses a key, this generates an electronic signal that is interpreted by the transducer in the keyboard and translated into the bit pattern of the corresponding IRA code. This bit pattern is then transmitted to the I/O module in the computer. At the computer, the text can be stored in the same IRA code. On output, IRA code characters are transmitted to an external device from the I/O module. The transducer at the device interprets this code and sends the required electronic signals to the output device either to display the indicated character or perform the requested control function.

Disk Drive

A disk drive contains electronics for exchanging data, control, and status signals with an I/O module plus the electronics for controlling the disk read/write mechanism. In a fixed-head disk, the transducer is capable of converting between the magnetic patterns on the moving disk surface and bits in the device's buffer (Figure 7.2). A moving-head disk must also be able to cause the disk arm to move radially in and out across the disk's surface.

7.2 I/O MODULES

Module Function

The major functions or requirements for an I/O module fall into the following categories:

- Control and timing
- Processor communication
- Device communication
- Data buffering
- Error detection

During any period of time, the processor may communicate with one or more external devices in unpredictable patterns, depending on the program's need for

[1]IRA is defined in ITU-T Recommendation T.50 and was formerly known as International Alphabet Number 5 (IA5). The U.S. national version of IRA is referred to as the American Standard Code for Information Interchange (ASCII).

I/O. The internal resources, such as main memory and the system bus, must be shared among a number of activities, including data I/O. Thus, the I/O function includes a **control and timing** requirement, to coordinate the flow of traffic between internal resources and external devices. For example, the control of the transfer of data from an external device to the processor might involve the following sequence of steps:

1. The processor interrogates the I/O module to check the status of the attached device.
2. The I/O module returns the device status.
3. If the device is operational and ready to transmit, the processor requests the transfer of data, by means of a command to the I/O module.
4. The I/O module obtains a unit of data (e.g., 8 or 16 bits) from the external device.
5. The data are transferred from the I/O module to the processor.

If the system employs a bus, then each of the interactions between the processor and the I/O module involves one or more bus arbitrations.

The preceding simplified scenario also illustrates that the I/O module must communicate with the processor and with the external device. **Processor communication** involves the following:

- **Command decoding:** The I/O module accepts commands from the processor, typically sent as signals on the control bus. For example, an I/O module for a disk drive might accept the following commands: READ SECTOR, WRITE SECTOR, SEEK track number, and SCAN record ID. The latter two commands each include a parameter that is sent on the data bus.
- **Data:** Data are exchanged between the processor and the I/O module over the data bus.
- **Status reporting:** Because peripherals are so slow, it is important to know the status of the I/O module. For example, if an I/O module is asked to send data to the processor (read), it may not be ready to do so because it is still working on the previous I/O command. This fact can be reported with a status signal. Common status signals are BUSY and READY. There may also be signals to report various error conditions.
- **Address recognition:** Just as each word of memory has an address, so does each I/O device. Thus, an I/O module must recognize one unique address for each peripheral it controls.

On the other side, the I/O module must be able to perform **device communication**. This communication involves commands, status information, and data (Figure 7.2).

An essential task of an I/O module is **data buffering**. The need for this function is apparent from Figure 2.1. Whereas the transfer rate into and out of main memory or the processor is quite high, the rate is orders of magnitude lower for many peripheral devices and covers a wide range. Data coming from main memory are sent to an I/O module in a rapid burst. The data are buffered in the I/O module and then sent to the peripheral device at its data rate. In the opposite direction, data are buffered so as not to tie up the memory in a slow transfer operation. Thus, the

I/O module must be able to operate at both device and memory speeds. Similarly, if the I/O device operates at a rate higher than the memory access rate, then the I/O module performs the needed buffering operation.

Finally, an I/O module is often responsible for **error detection** and for subsequently reporting errors to the processor. One class of errors includes mechanical and electrical malfunctions reported by the device (e.g., paper jam, bad disk track). Another class consists of unintentional changes to the bit pattern as it is transmitted from device to I/O module. Some form of error-detecting code is often used to detect transmission errors. A simple example is the use of a parity bit on each character of data. For example, the IRA character code occupies 7 bits of a byte. The eighth bit is set so that the total number of 1s in the byte is even (even parity) or odd (odd parity). When a byte is received, the I/O module checks the parity to determine whether an error has occurred.

I/O Module Structure

I/O modules vary considerably in complexity and the number of external devices that they control. We will attempt only a very general description here. (One specific device, the Intel 8255A, is described in Section 7.4.) Figure 7.3 provides a general block diagram of an I/O module. The module connects to the rest of the computer through a set of signal lines (e.g., system bus lines). Data transferred to and from the module are buffered in one or more data registers. There may also be one or more status registers that provide current status information. A status register may also function as a control register, to accept detailed control information from the processor. The logic within the module interacts with the processor via a set of control lines. The processor uses the control lines to issue commands

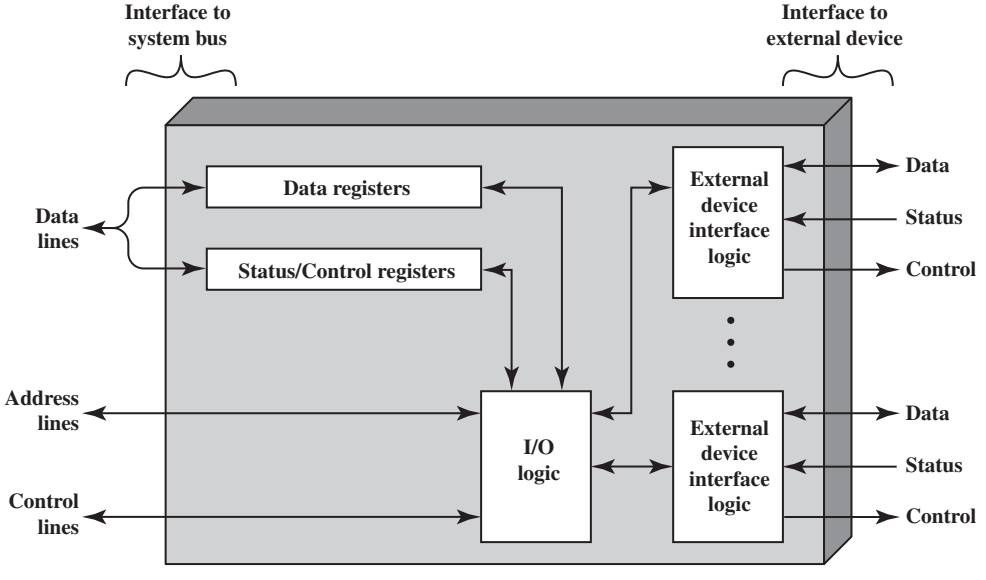


Figure 7.3 Block Diagram of an I/O Module

to the I/O module. Some of the control lines may be used by the I/O module (e.g., for arbitration and status signals). The module must also be able to recognize and generate addresses associated with the devices it controls. Each I/O module has a unique address or, if it controls more than one external device, a unique set of addresses. Finally, the I/O module contains logic specific to the interface with each device that it controls.

An I/O module functions to allow the processor to view a wide range of devices in a simple-minded way. There is a spectrum of capabilities that may be provided. The I/O module may hide the details of timing, formats, and the electromechanics of an external device so that the processor can function in terms of simple read and write commands, and possibly open and close file commands. In its simplest form, the I/O module may still leave much of the work of controlling a device (e.g., rewind a tape) visible to the processor.

An I/O module that takes on most of the detailed processing burden, presenting a high-level interface to the processor, is usually referred to as an I/O channel or *I/O processor*. An I/O module that is quite primitive and requires detailed control is usually referred to as an *I/O controller* or *device controller*. I/O controllers are commonly seen on microcomputers, whereas I/O channels are used on mainframes.

In what follows, we will use the generic term *I/O module* when no confusion results and will use more specific terms where necessary.

7.3 PROGRAMMED I/O

Three techniques are possible for I/O operations. With *programmed I/O*, data are exchanged between the processor and the I/O module. The processor executes a program that gives it direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data. When the processor issues a command to the I/O module, it must wait until the I/O operation is complete. If the processor is faster than the I/O module, this is waste of processor time. With interrupt-driven I/O, the processor issues an *I/O command*, continues to execute other instructions, and is interrupted by the I/O module when the latter has completed its work. With both programmed and *interrupt* I/O, the processor is responsible for extracting data from main memory for output and storing data in main memory for input. The alternative is known as **direct memory access (DMA)**. In this mode, the I/O module and main memory exchange data directly, without processor involvement.

Table 7.1 indicates the relationship among these three techniques. In this section, we explore programmed I/O. Interrupt I/O and DMA are explored in the following two sections, respectively.

Table 7.1 I/O Techniques

	No Interrupts	Use of Interrupts
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

Overview of Programmed I/O

When the processor is executing a program and encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module. With programmed I/O, the I/O module will perform the requested action and then set the appropriate bits in the I/O status register (Figure 7.3). The I/O module takes no further action to alert the processor. In particular, it does not interrupt the processor. Thus, it is the responsibility of the processor to periodically check the status of the I/O module until it finds that the operation is complete.

To explain the programmed I/O technique, we view it first from the point of view of the I/O commands issued by the processor to the I/O module, and then from the point of view of the I/O instructions executed by the processor.

I/O Commands

To execute an I/O-related instruction, the processor issues an address, specifying the particular I/O module and external device, and an I/O command. There are four types of I/O commands that an I/O module may receive when it is addressed by a processor:

- **Control:** Used to activate a peripheral and tell it what to do. For example, a magnetic-tape unit may be instructed to rewind or to move forward one record. These commands are tailored to the particular type of peripheral device.
- **Test:** Used to test various status conditions associated with an **I/O module** and its peripherals. The processor will want to know that the peripheral of interest is powered on and available for use. It will also want to know if the most recent I/O operation is completed and if any errors occurred.
- **Read:** Causes the I/O module to obtain an item of data from the peripheral and place it in an internal buffer (depicted as a data register in Figure 7.3). The processor can then obtain the data item by requesting that the I/O module place it on the data bus.
- **Write:** Causes the I/O module to take an item of data (byte or word) from the data bus and subsequently transmit that data item to the peripheral.

Figure 7.4a gives an example of the use of programmed I/O to read in a block of data from a peripheral device (e.g., a record from tape) into memory. Data are read in one word (e.g., 16 bits) at a time. For each word that is read in, the processor must remain in a status-checking cycle until it determines that the word is available in the I/O module's data register. This flowchart highlights the main disadvantage of this technique: it is a time-consuming process that keeps the processor busy needlessly.

I/O Instructions

With programmed I/O, there is a close correspondence between the I/O-related instructions that the processor fetches from memory and the I/O commands that the processor issues to an I/O module to execute the instructions. That is, the instructions are easily mapped into I/O commands, and there is often a simple one-to-one relationship. The form of the instruction depends on the way in which external devices are addressed.

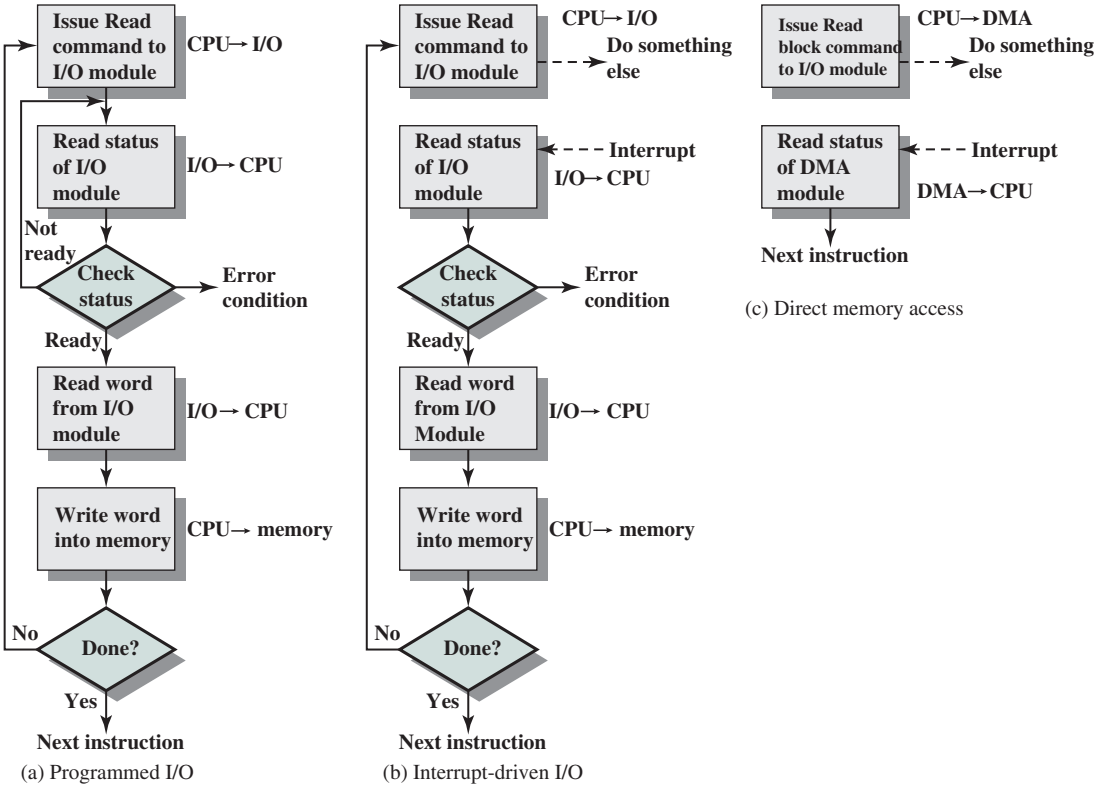


Figure 7.4 Three Techniques for Input of a Block of Data

Typically, there will be many I/O devices connected through I/O modules to the system. Each device is given a unique identifier or address. When the processor issues an I/O command, the command contains the address of the desired device. Thus, each I/O module must interpret the address lines to determine if the command is for itself.

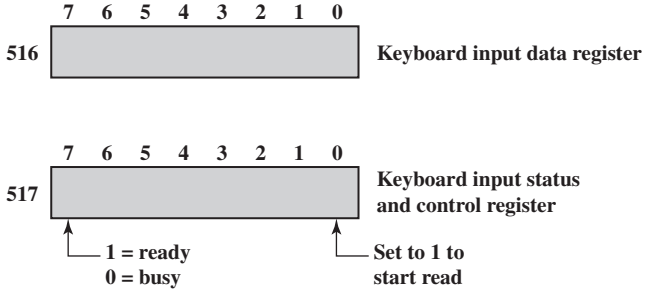
When the processor, main memory, and I/O share a common bus, two modes of addressing are possible: memory mapped and isolated. With **memory-mapped I/O**, there is a single address space for memory locations and I/O devices. The processor treats the status and data registers of I/O modules as memory locations and uses the same machine instructions to access both memory and I/O devices. So, for example, with 10 address lines, a combined total of $2^{10} = 1024$ memory locations and I/O addresses can be supported, in any combination.

With memory-mapped I/O, a single read line and a single write line are needed on the bus. Alternatively, the bus may be equipped with memory read and write plus input and output command lines. The command line specifies whether the address refers to a memory location or an I/O device. The full range of addresses may be available for both. Again, with 10 address lines, the system may now support both 1024 memory locations and 1024 I/O addresses. Because the address space for I/O is isolated from that for memory, this is referred to as **isolated I/O**.

Figure 7.5 contrasts these two programmed I/O techniques. Figure 7.5a shows how the interface for a simple input device such as a terminal keyboard might appear to a programmer using memory-mapped I/O. Assume a 10-bit address, with a 512-bit memory (locations 0–511) and up to 512 I/O addresses (locations 512–1023). Two addresses are dedicated to keyboard input from a particular terminal. Address 516 refers to the data register and address 517 refers to the status register, which also functions as a control register for receiving processor commands. The program shown will read 1 byte of data from the keyboard into an accumulator register in the processor. Note that the processor loops until the data byte is available.

With isolated I/O (Figure 7.5b), the I/O ports are accessible only by special I/O commands, which activate the I/O command lines on the bus.

For most types of processors, there is a relatively large set of different instructions for referencing memory. If isolated I/O is used, there are only a few I/O instructions. Thus, an advantage of memory-mapped I/O is that this large repertoire of instructions can be used, allowing more efficient programming. A disadvantage is that valuable memory address space is used up. Both memory-mapped and isolated I/O are in common use.



ADDRESS	INSTRUCTION	OPERAND	COMMENT
200	Load AC	"1"	Load accumulator
	Store AC	517	Initiate keyboard read
202	Load AC	517	Get status byte
	Branch if Sign = 0	202	Loop until ready
	Load AC	516	Load data byte

(a) Memory-mapped I/O

ADDRESS	INSTRUCTION	OPERAND	COMMENT
200	Load I/O	5	Initiate keyboard read
201	Test I/O	5	Check for completion
	Branch Not Ready	201	Loop until complete
	In	5	Load data byte

(b) Isolated I/O

Figure 7.5 Memory-Mapped and Isolated I/O

7.4 INTERRUPT-DRIVEN I/O

The problem with programmed I/O is that the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of data. The processor, while waiting, must repeatedly interrogate the status of the I/O module. As a result, the level of the performance of the entire system is severely degraded.

An alternative is for the processor to issue an I/O command to a module and then go on to do some other useful work. The I/O module will then interrupt the processor to request service when it is ready to exchange data with the processor. The processor then executes the data transfer, as before, and then resumes its former processing.

Let us consider how this works, first from the point of view of the I/O module. For input, the I/O module receives a READ command from the processor. The I/O module then proceeds to read data in from an associated peripheral. Once the data are in the module's data register, the module signals an interrupt to the processor over a control line. The module then waits until its data are requested by the processor. When the request is made, the module places its data on the data bus and is then ready for another I/O operation.

From the processor's point of view, the action for input is as follows. The processor issues a READ command. It then goes off and does something else (e.g., the processor may be working on several different programs at the same time). At the end of each instruction cycle, the processor checks for interrupts (Figure 3.9). When the interrupt from the I/O module occurs, the processor saves the context (e.g., program counter and processor registers) of the current program and processes the interrupt. In this case, the processor reads the word of data from the I/O module and stores it in memory. It then restores the context of the program it was working on (or some other program) and resumes execution.

Figure 7.4b shows the use of interrupt I/O for reading in a block of data. Compare this with Figure 7.4a. Interrupt I/O is more efficient than programmed I/O because it eliminates needless waiting. However, interrupt I/O still consumes a lot of processor time, because every word of data that goes from memory to I/O module or from I/O module to memory must pass through the processor.

Interrupt Processing

Let us consider the role of the processor in interrupt-driven I/O in more detail. The occurrence of an interrupt triggers a number of events, both in the processor hardware and in software. Figure 7.6 shows a typical sequence. When an I/O device completes an I/O operation, the following sequence of hardware events occurs:

1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt, as indicated in Figure 3.9.
3. The processor tests for an interrupt, determines that there is one, and sends an acknowledgment signal to the device that issued the interrupt. The acknowledgment allows the device to remove its interrupt signal.

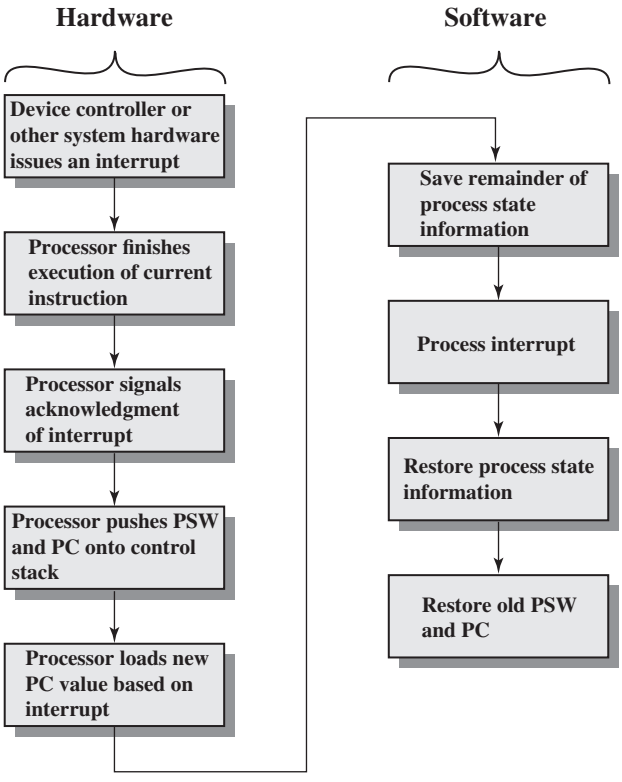

Hardware
Software
Device controller or other system hardware issues an interrupt
Processor finishes execution of current instruction
Processor signals acknowledgment of interrupt
Processor pushes PSW and PC onto control stack
Processor loads new PC value based on interrupt
Save remainder of process state information
Process interrupt
Restore process state information
Restore old PSW and PC

Figure 7.6 Simple Interrupt Processing

4. The processor now needs to prepare to transfer control to the interrupt routine. To begin, it needs to save information needed to resume the current program at the point of interrupt. The minimum information required is (a) the status of the processor, which is contained in a register called the **program status word (PSW)**; and (b) the location of the next instruction to be executed, which is contained in the program counter. These can be pushed onto the system control stack.[2]
5. The processor now loads the program counter with the entry location of the interrupt-handling program that will respond to this interrupt. Depending on the computer architecture and operating system design, there may be a single program; one program for each type of interrupt; or one program for each device and each type of interrupt. If there is more than one interrupt-handling routine, the processor must determine which one to invoke. This information may have been included in the original interrupt signal, or the processor may have to issue a request to the device that issued the interrupt to get a response that contains the needed information.

[2]See Appendix I for a discussion of stack operation.

Once the program counter has been loaded, the processor proceeds to the next instruction cycle, which begins with an instruction fetch. Because the instruction fetch is determined by the contents of the program counter, the result is that control is transferred to the interrupt-handler program. The execution of this program results in the following operations:

6. At this point, the program counter and PSW relating to the interrupted program have been saved on the system stack. However, there is other information that is considered part of the "state" of the executing program. In particular, the contents of the processor registers need to be saved, because these registers may be used by the interrupt handler. So, all of these values, plus any other state information, need to be saved. Typically, the interrupt handler will begin by saving the contents of all registers on the stack. Figure 7.7a shows a simple example. In this case, a user program is interrupted after the instruction at location N. The contents of all of the registers plus the address of the next instruction ($N + 1$) are pushed onto the stack. The stack pointer is updated to point to the new top of stack, and the program counter is updated to point to the beginning of the interrupt service routine.
7. The interrupt handler next processes the interrupt. This includes an examination of status information relating to the I/O operation or other event that caused an interrupt. It may also involve sending additional commands or acknowledgments to the I/O device.
8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers (e.g., see Figure 7.7b).
9. The final act is to restore the PSW and program counter values from the stack. As a result, the next instruction to be executed will be from the previously interrupted program.

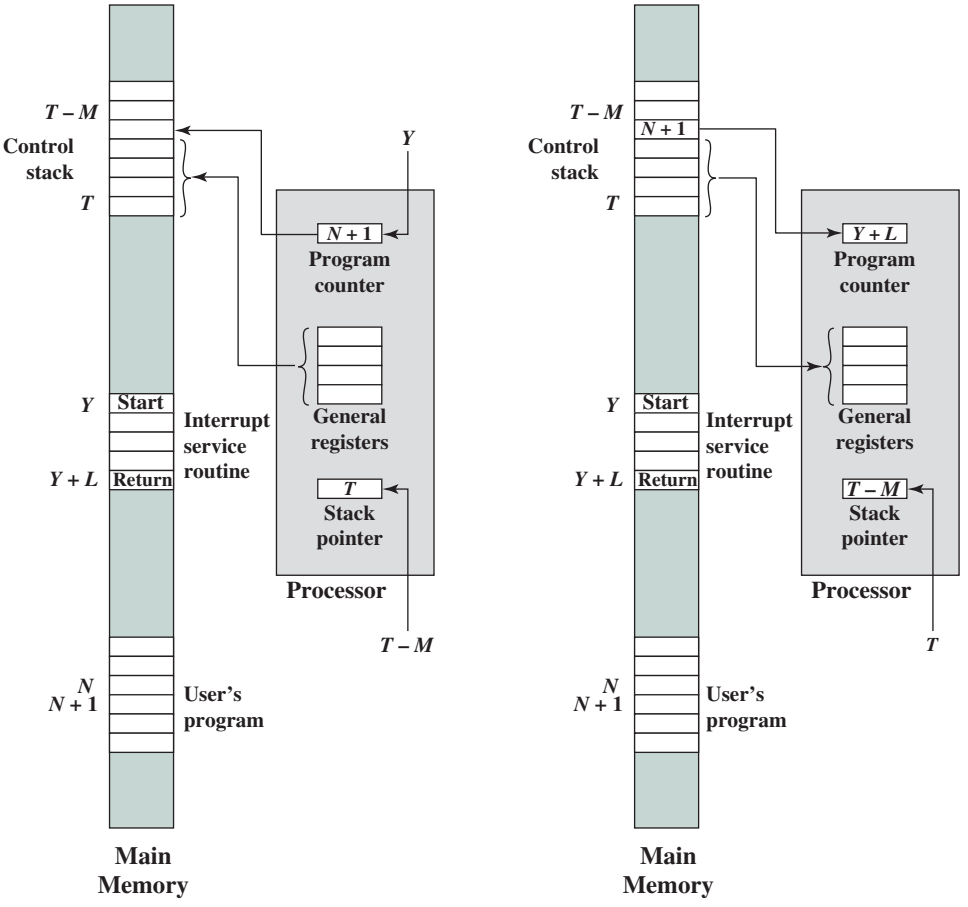
Note that it is important to save all the state information about the interrupted program for later resumption. This is because the interrupt is not a routine called from the program. Rather, the interrupt can occur at any time and therefore at any point in the execution of a user program. Its occurrence is unpredictable. Indeed, as we will see in the next chapter, the two programs may not have anything in common and may belong to two different users.

Design Issues

Two design issues arise in implementing interrupt I/O. First, because there will almost invariably be multiple I/O modules, how does the processor determine which device issued the interrupt? And second, if multiple interrupts have occurred, how does the processor decide which one to process?

Let us consider device identification first. Four general categories of techniques are in common use:

- Multiple interrupt lines
- Software poll
- Daisy chain (hardware poll, vectored)
- Bus arbitration (vectored)



(a) Interrupt occurs after instruction at location *N*

(b) Return from interrupt

Figure 7.7 Changes in Memory and Registers for an Interrupt

The most straightforward approach to the problem is to provide **multiple interrupt lines** between the processor and the I/O modules. However, it is impractical to dedicate more than a few bus lines or processor pins to interrupt lines. Consequently, even if multiple lines are used, it is likely that each line will have multiple I/O modules attached to it. Thus, one of the other three techniques must be used on each line.

One alternative is the **software poll**. When the processor detects an interrupt, it branches to an interrupt-service routine that polls each I/O module to determine which module caused the interrupt. The poll could be in the form of a separate command line (e.g., TESTI/O). In this case, the processor raises TESTI/O and places the address of a particular I/O module on the address lines. The I/O module responds positively if it set the interrupt. Alternatively, each I/O module could contain an addressable status register. The processor then reads the status register of each I/O module to identify the interrupting module. Once the correct module is identified, the processor branches to a device-service routine specific to that device.

The disadvantage of the software poll is that it is time consuming. A more efficient technique is to use a **daisy chain**, which provides, in effect, a hardware poll. An example of a daisy-chain configuration is shown in Figure 3.26. For interrupts, all I/O modules share a common interrupt request line. The interrupt acknowledge line is daisy chained through the modules. When the processor senses an interrupt, it sends out an interrupt acknowledge. This signal propagates through a series of I/O modules until it gets to a requesting module. The requesting module typically responds by placing a word on the data lines. This word is referred to as a *vector* and is either the address of the I/O module or some other unique identifier. In either case, the processor uses the vector as a pointer to the appropriate device-service routine. This avoids the need to execute a general interrupt-service routine first. This technique is called a *vectored interrupt.*

There is another technique that makes use of vectored interrupts, and that is **bus arbitration**. With bus arbitration, an I/O module must first gain control of the bus before it can raise the interrupt request line. Thus, only one module can raise the line at a time. When the processor detects the interrupt, it responds on the interrupt acknowledge line. The requesting module then places its vector on the data lines.

The aforementioned techniques serve to identify the requesting I/O module. They also provide a way of assigning priorities when more than one device is requesting interrupt service. With multiple lines, the processor just picks the interrupt line with the highest priority. With software polling, the order in which modules are polled determines their priority. Similarly, the order of modules on a daisy chain determines their priority. Finally, bus arbitration can employ a priority scheme, as discussed in Section 3.4.

We now turn to two examples of interrupt structures.

Intel 82C59A Interrupt Controller

The Intel 80386 provides a single Interrupt Request (INTR) and a single Interrupt Acknowledge (INTA) line. To allow the 80386 to handle a variety of devices and priority structures, it is usually configured with an external interrupt arbiter, the 82C59A. External devices are connected to the 82C59A, which in turn connects to the 80386.

Figure 7.8 shows the use of the 82C59A to connect multiple I/O modules for the 80386. A single 82C59A can handle up to eight modules. If control for more than eight modules is required, a cascade arrangement can be used to handle up to 64 modules.

The 82C59A's sole responsibility is the management of interrupts. It accepts interrupt requests from attached modules, determines which interrupt has the highest priority, and then signals the processor by raising the INTR line. The processor acknowledges via the INTA line. This prompts the 82C59A to place the appropriate vector information on the data bus. The processor can then proceed to process the interrupt and to communicate directly with the I/O module to read or write data.

The 82C59A is programmable. The 80386 determines the priority scheme to be used by setting a control word in the 82C59A. The following interrupt modes are possible:

- **Fully nested:** The interrupt requests are ordered in priority from 0 (IR0) through 7 (IR7).

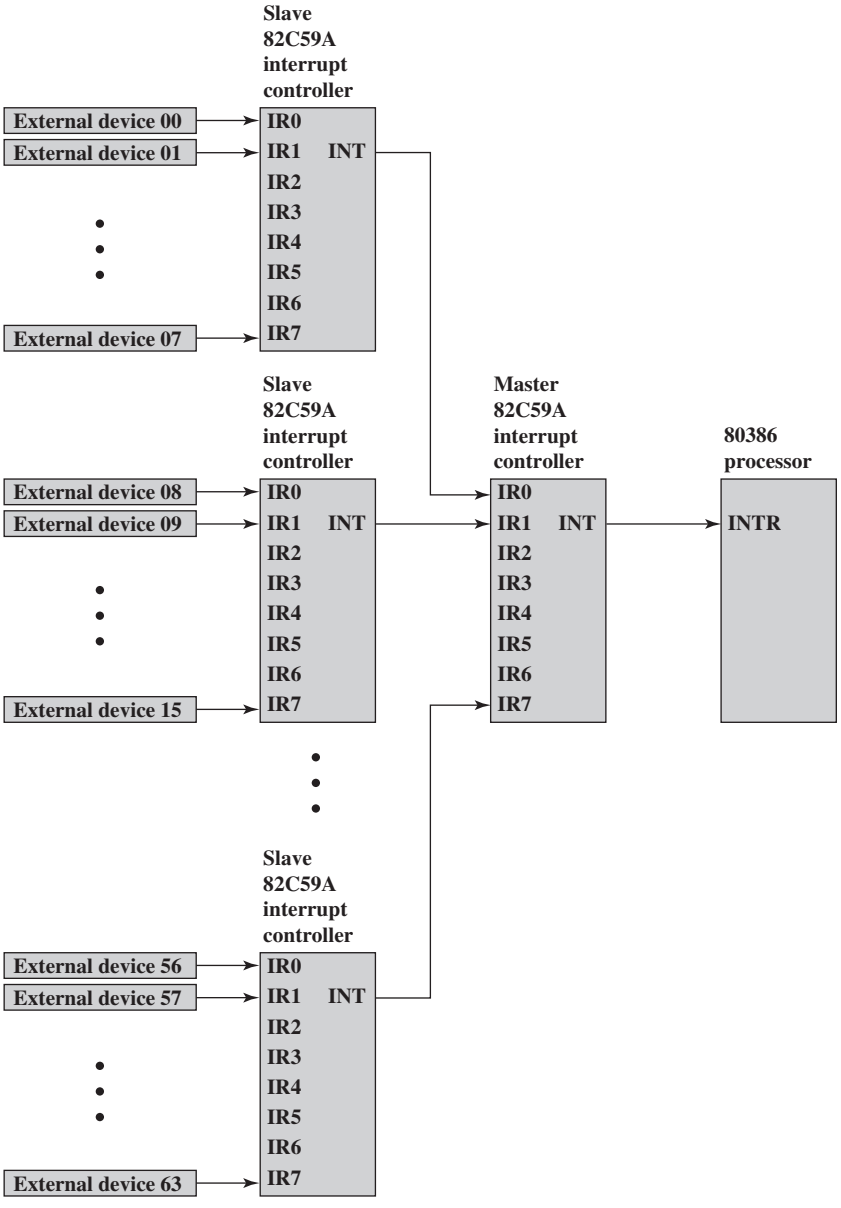


Figure 7.8 Use of the 82C59A Interrupt Controller

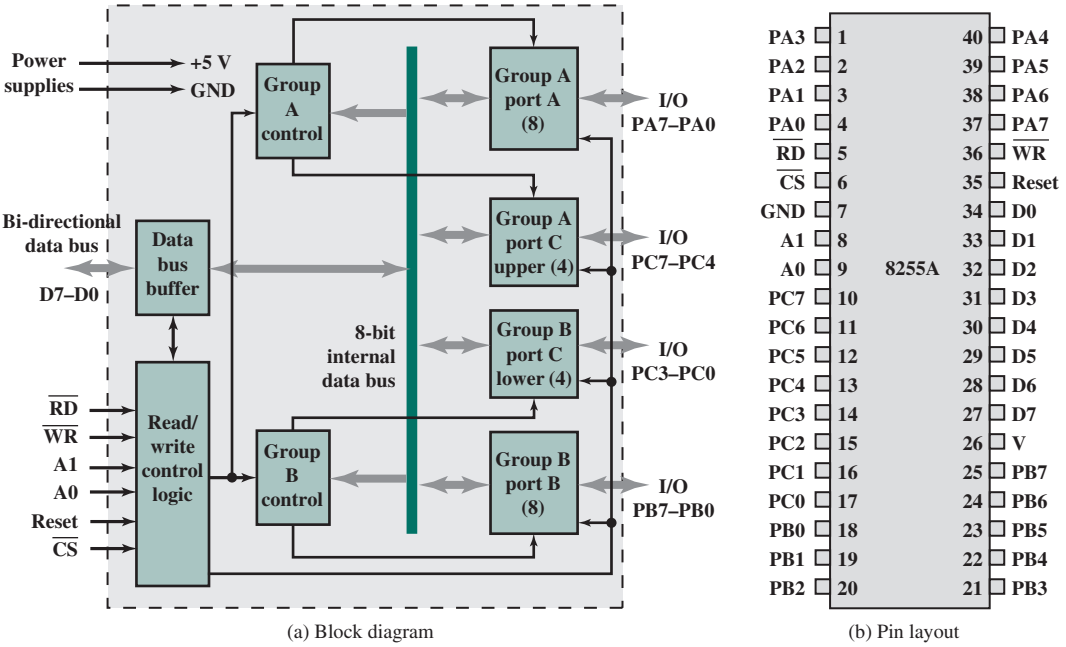
- **Rotating:** In some applications a number of interrupting devices are of equal priority. In this mode a device, after being serviced, receives the lowest priority in the group.
- **Special mask:** This allows the processor to inhibit interrupts from certain devices.

The Intel 8255A Programmable Peripheral Interface

As an example of an I/O module used for programmed I/O and interrupt-driven I/O, we consider the Intel 8255A Programmable Peripheral Interface. The 8255A is a single-chip, general-purpose I/O module originally designed for use with the Intel 80386 processor. It has since been cloned by other manufacturers and is a widely used peripheral controller chip. Its uses include as a controller for simple I/O devices for microprocessors and in embedded systems, including microcontroller systems.

ARCHITECTURE AND OPERATION Figure 7.9 shows a general block diagram plus the pin assignment for the 40-pin package in which it is housed. As shown on the pin layout, the 8255A includes the following lines:

- **D0–D7:** These are the data I/O lines for the device. All information read from and written to the 8255A occurs via these eight data lines.
- **$\overline{\text{CS}}$ (Chip Select Input):** If this line is a logical 0, the microprocessor can read and write to the 8255A.
- **$\overline{\text{RD}}$ (Read Input):** If this line is a logical 0 and the $\overline{\text{CS}}$ input is a logical 0, the 8255A data outputs are enabled onto the system data bus.
- **$\overline{\text{WR}}$ (Write Input):** If this input line is a logical 0 and the $\overline{\text{CS}}$ input is a logical 0, data are written to the 8255A from the system data bus.
- **RESET:** The 8255A is placed into its reset state if this input line is a logical 1. All peripheral ports are set to the input mode.



(a) Block diagram (b) Pin layout

Figure 7.9 The Intel 8255A Programmable Peripheral Interface

- **PA0–PA7, PB0–PB7, PC0–PC7:** These signal lines are used as 8-bit I/O ports. They can be connected to peripheral devices.
- **A0, A1:** The logical combination of these two input lines determine which internal register of the 8255A data are written to or read from.

The right side of the block diagram of Figure 7.9a is the external interface of the 8255A. The 24 I/O lines are divided into three 8-bit groups (A, B, C). Each group can function as an 8-bit I/O port, thus providing connection for three peripheral devices. In addition, group C is subdivided into 4-bit groups (C_A and C_B), which may be used in conjunction with the A and B I/O ports. Configured in this manner, group C lines carry control and status signals.

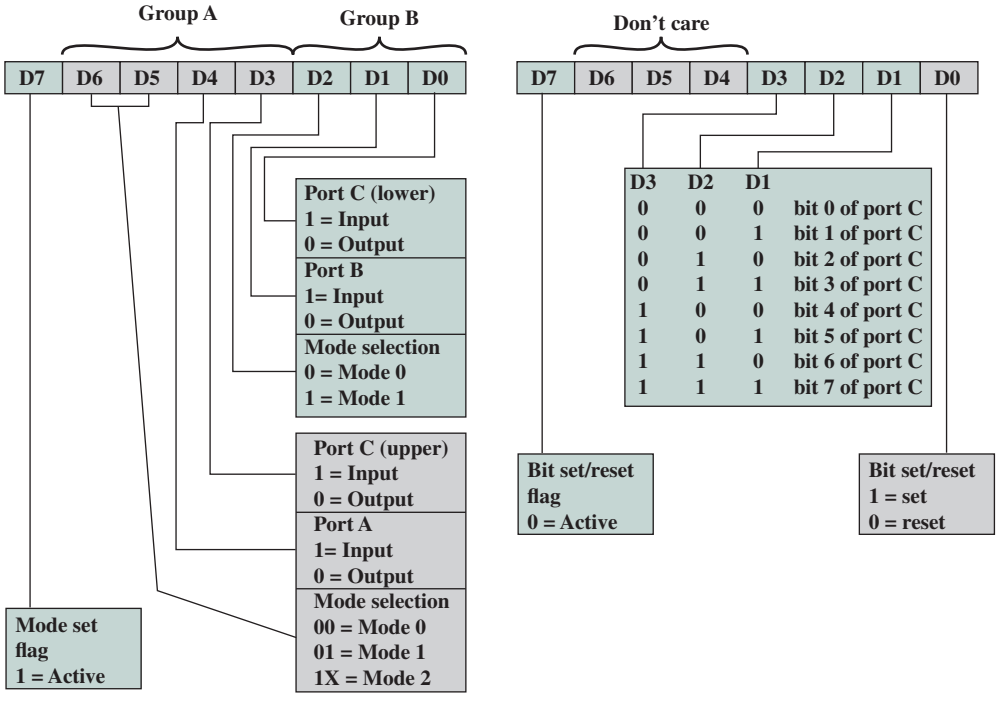
The left side of the block diagram is the internal interface to the microprocessor system bus. It includes an 8-bit bidirectional data bus (D0 through D7), used to transfer data between the microprocessor and the I/O ports and to transfer control information.

The processor controls the 8255A by means of an 8-bit control register in the processor. The processor can set the value of the control register to specify a variety of operating modes and configurations. From the processor point of view, there is a control port, and the control register bits are set in the processor and then sent to the control port over lines D0–D7. The two address lines specify one of the three I/O ports or the control register, as follows:

A1	A2	Selects
0	0	Port A
0	1	Port B
1	0	Port C
1	1	Control register

Thus, when the processor sets both A1 and A2 to 1, the 8255A interprets the 8-bit value on the data bus as a control word. When the processor transfers an 8-bit control word with line D7 set to 1 (Figure 7.10a), the control word is used to configure the operating mode of the 24 I/O lines. The three modes are:

- **Mode 0:** This is the basic I/O mode. The three groups of eight external lines function as three 8-bit I/O ports. Each port can be designated as input or output. Data may only be sent to a port if the port is defined as output, and data may only be read from a port if the port is set to input.
- **Mode 1:** In this mode, ports A and B can be configured as either input or output, and lines from port C serve as control lines for A and B. The control signals serve two principal purposes: "handshaking" and interrupt request. Handshaking is a simple timing mechanism. One control line is used by the sender as a DATA READY line, to indicate when the data are present on the I/O data lines. Another line is used by the receiver as an ACKNOWLEDGE, indicating that the data have been read and the data lines may be cleared. Another line may be designated as an INTERRUPT REQUEST line and tied back to the system bus.



(a) Mode definition of the 8255 control register to configure the 8255

(b) Bit definitions of the 8255 control register to modify single bits of port C

Figure 7.10 The Intel 8255A Control Word

- **Mode 2:** This is a bidirectional mode. In this mode, port A can be configured as either the input or output lines for bidirectional traffic on port B, with the port B lines providing the opposite direction. Again, port C lines are used for control signaling.

When the processor sets D7 to 0 (Figure 7.10b), the control word is used to program the bit values of port C individually. This feature is rarely used.

KEYBOARD/DISPLAY EXAMPLE Because the 8255A is programmable via the control register, it can be used to control a variety of simple peripheral devices. Figure 7.11 illustrates its use to control a keyboard/display terminal. The keyboard provides 8 bits of input. Two of these bits, SHIFT and CONTROL, have special meaning to the keyboard-handling program executing in the processor. However, this interpretation is transparent to the 8255A, which simply accepts the 8 bits of data and presents them on the system data bus. Two handshaking control lines are provided for use with the keyboard.

The display is also linked by an 8-bit data port. Again, two of the bits have special meanings that are transparent to the 8255A. In addition to two handshaking lines, two lines provide additional control functions.

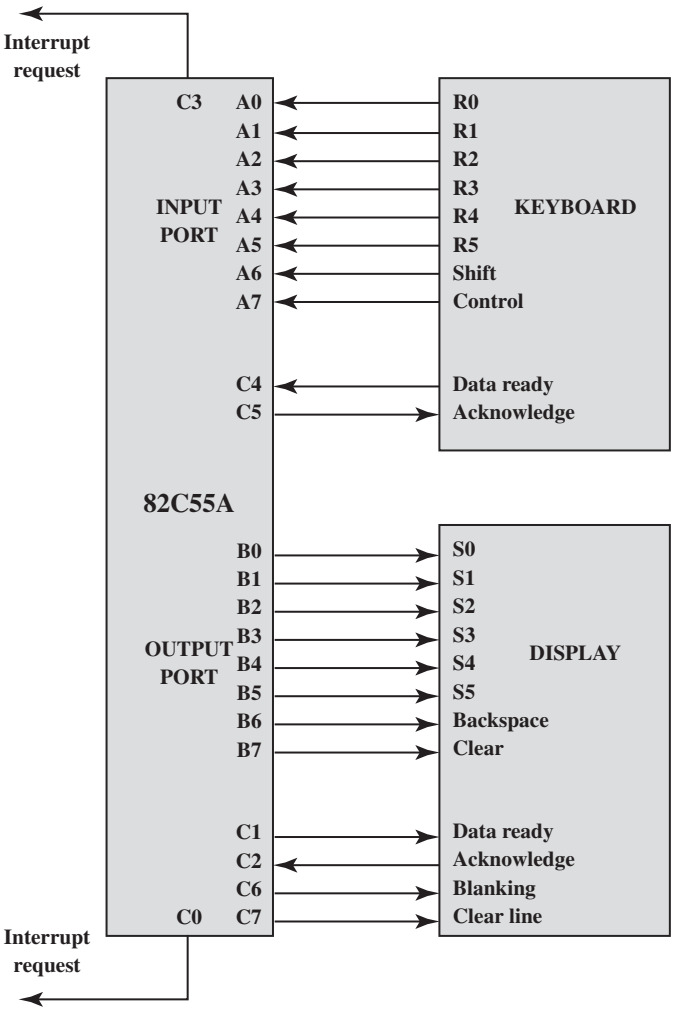


Figure 7.11 Keyboard/Display Interface to 8255A

7.5 DIRECT MEMORY ACCESS

Drawbacks of Programmed and Interrupt-Driven I/O

Interrupt-driven I/O, though more efficient than simple programmed I/O, still requires the active intervention of the processor to transfer data between memory and an I/O module, and any data transfer must traverse a path through the processor. Thus, both these forms of I/O suffer from two inherent drawbacks:

1. The I/O transfer rate is limited by the speed with which the processor can test and service a device.

2. The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer (e.g., Figure 7.5).

There is somewhat of a trade-off between these two drawbacks. Consider the transfer of a block of data. Using simple programmed I/O, the processor is dedicated to the task of I/O and can move data at a rather high rate, at the cost of doing nothing else. Interrupt I/O frees up the processor to some extent at the expense of the I/O transfer rate. Nevertheless, both methods have an adverse impact on both processor activity and I/O transfer rate.

When large volumes of data are to be moved, a more efficient technique is required: direct memory access (DMA).

DMA Function

DMA involves an additional module on the system bus. The DMA module (Figure 7.12) is capable of mimicking the processor and, indeed, of taking over control of the system from the processor. It needs to do this to transfer data to and from memory over the system bus. For this purpose, the DMA module must use the bus only when the processor does not need it, or it must force the processor to suspend operation temporarily. The latter technique is more common and is referred to as *cycle stealing*, because the DMA module in effect steals a bus cycle.

When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information:

- Whether a read or write is requested, using the read or write control line between the processor and the DMA module.
- The address of the I/O device involved, communicated on the data lines.

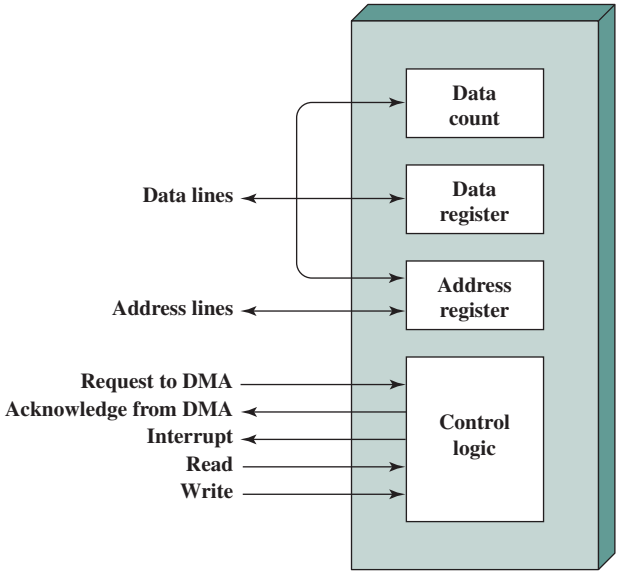


Figure 7.12 Typical DMA Block Diagram

- The starting location in memory to read from or write to, communicated on the data lines and stored by the DMA module in its address register.
- The number of words to be read or written, again communicated via the data lines and stored in the data count register.

The processor then continues with other work. It has delegated this I/O operation to the DMA module. The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor. Thus, the processor is involved only at the beginning and end of the transfer (Figure 7.4c).

Figure 7.13 shows where in the instruction cycle the processor may be suspended. In each case, the processor is suspended just before it needs to use the bus. The DMA module then transfers one word and returns control to the processor. Note that this is not an interrupt; the processor does not save a context and do something else. Rather, the processor pauses for one bus cycle. The overall effect is to cause the processor to execute more slowly. Nevertheless, for a multiple-word I/O transfer, DMA is far more efficient than interrupt-driven or programmed I/O.

The DMA mechanism can be configured in a variety of ways. Some possibilities are shown in Figure 7.14. In the first example, all modules share the same system bus. The DMA module, acting as a surrogate processor, uses programmed I/O to exchange data between memory and an I/O module through the DMA module. This configuration, while it may be inexpensive, is clearly inefficient. As with processor-controlled programmed I/O, each transfer of a word consumes two bus cycles.

The number of required bus cycles can be cut substantially by integrating the DMA and I/O functions. As Figure 7.14b indicates, this means that there is a path between the DMA module and one or more I/O modules that does not include

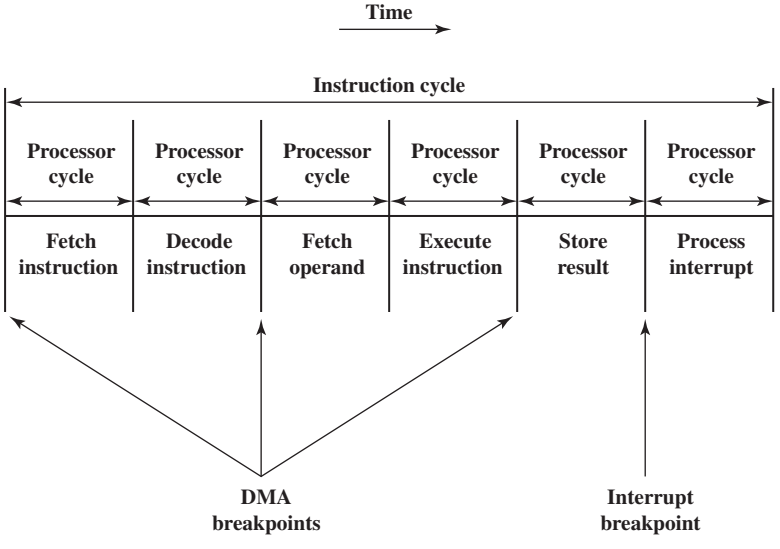


Figure 7.13 DMA and Interrupt Breakpoints during an Instruction Cycle

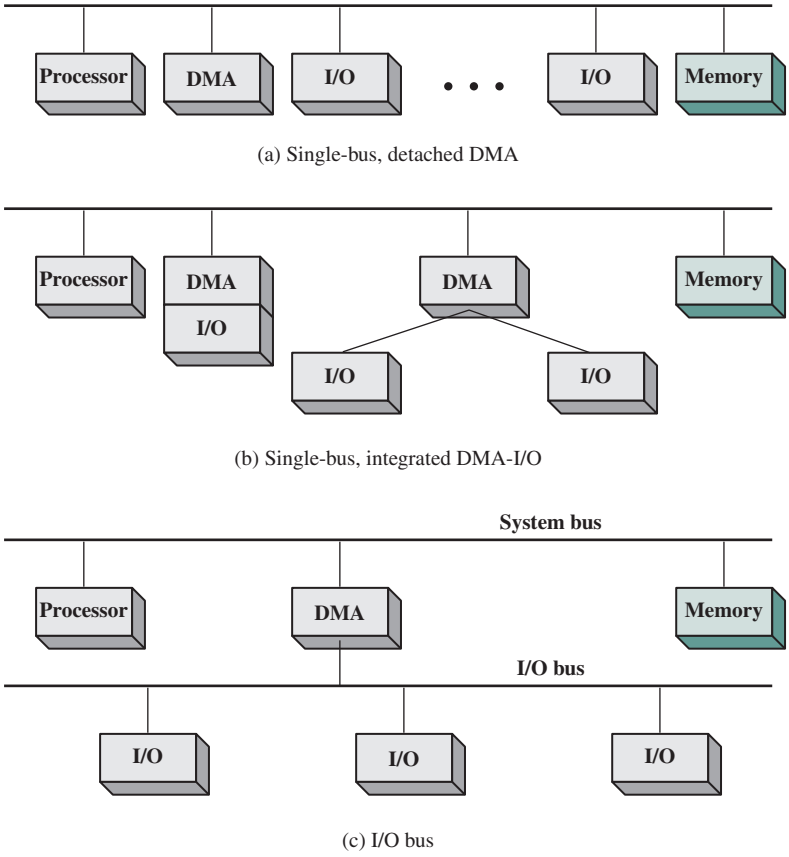


Figure 7.14 Alternative DMA Configurations

the system bus. The DMA logic may actually be a part of an I/O module, or it may be a separate module that controls one or more I/O modules. This concept can be taken one step further by connecting I/O modules to the DMA module using an I/O bus (Figure 7.14c). This reduces the number of I/O interfaces in the DMA module to one and provides for an easily expandable configuration. In both of these cases (Figures 7.14b and c), the system bus that the DMA module shares with the processor and memory is used by the DMA module only to exchange data with memory. The exchange of data between the DMA and I/O modules takes place off the system bus.

Intel 8237A DMA Controller

The Intel 8237A DMA controller interfaces to the 80 × 86 family of processors and to DRAM memory to provide a DMA capability. Figure 7.15 indicates the location of the DMA module. When the DMA module needs to use the system buses (data, address, and control) to transfer data, it sends a signal called HOLD to the processor. The processor responds with the HLDA (hold acknowledge) signal, indicating that

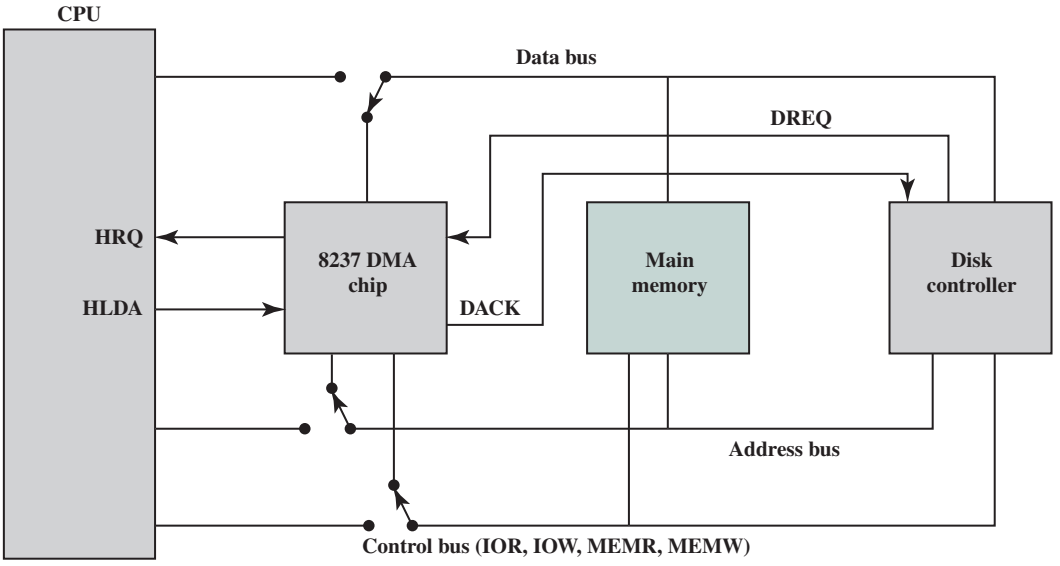


Figure 7.15 8237 DMA Usage of System Bus

the DMA module can use the buses. For example, if the DMA module is to transfer a block of data from memory to disk, it will do the following:

1. The peripheral device (such as the disk controller) will request the service of DMA by pulling DREQ (DMA request) high.
2. The DMA will put a high on its HRQ (hold request), signaling the CPU through its HOLD pin that it needs to use the buses.
3. The CPU will finish the present bus cycle (not necessarily the present instruction) and respond to the DMA request by putting high on its HDLA (hold acknowledge), thus telling the 8237 DMA that it can go ahead and use the buses to perform its task. HOLD must remain active high as long as DMA is performing its task.
4. DMA will activate DACK (DMA acknowledge), which tells the peripheral device that it will start to transfer the data.
5. DMA starts to transfer the data from memory to peripheral by putting the address of the first byte of the block on the address bus and activating MEMR, thereby reading the byte from memory into the data bus; it then activates IOW to write it to the peripheral. Then DMA decrements the counter and increments the address pointer and repeats this process until the count reaches zero and the task is finished.
6. After the DMA has finished its job it will deactivate HRQ, signaling the CPU that it can regain control over its buses.

While the DMA is using the buses to transfer data, the processor is idle. Similarly, when the processor is using the bus, the DMA is idle. The 8237 DMA is known as a *fly-by* DMA controller. This means that the data being moved from one location to another does not pass through the DMA chip and is not stored in the DMA chip. Therefore, the DMA can only transfer data between an I/O port and a memory address, and not between two I/O ports or two memory locations. However, as explained subsequently, the DMA chip can perform a memory-to-memory transfer via a register.

The 8237 contains four DMA channels that can be programmed independently, and any one of the channels may be active at any moment. These channels are numbered 0, 1, 2, and 3.

The 8237 has a set of five control/command registers to program and control DMA operation over one of its channels (Table 7.2):

- **Command:** The processor loads this register to control the operation of the DMA. D0 enables a memory-to-memory transfer, in which channel 0 is used to transfer a byte into an 8237 temporary register and channel 1 is used to transfer the byte from the register to memory. When memory-to-memory is enabled, D1 can be used to disable increment/decrement on channel 0 so that a fixed value can be written into a block of memory. D2 enables or disables DMA.
- **Status:** The processor reads this register to determine DMA status. Bits D0–D3 are used to indicate if channels 0–3 have reached their TC (terminal count). Bits D4–D7 are used by the processor to determine if any channel has a DMA request pending.
- **Mode:** The processor sets this register to determine the mode of operation of the DMA. Bits D0 and D1 are used to select a channel. The other bits select various operation modes for the selected channel. Bits D2 and D3 determine if the transfer is from an I/O device to memory (write) or from memory to I/O (read), or a verify operation. If D4 is set, then the memory address register and the count register are reloaded with their original values at the end of a DMA data transfer. Bits D6 and D7 determine the way in which the 8237 is used. In single mode, a single byte of data is transferred. Block and demand modes are used for a block transfer, with the demand mode allowing for premature ending of the transfer. Cascade mode allows multiple 8237s to be cascaded to expand the number of channels to more than 4.
- **Single Mask:** The processor sets this register. Bits D0 and D1 select the channel. Bit D2 clears or sets the mask bit for that channel. It is through this register that the DREQ input of a specific channel can be masked (disabled) or unmasked (enabled). While the command register can be used to disable the whole DMA chip, the single mask register allows the programmer to disable or enable a specific channel.
- **All Mask:** This register is similar to the single mask register except that all four channels can be masked or unmasked with one write operation.

In addition, the 8237A has eight data registers: one memory address register and one count register for each channel. The processor sets these registers to indicate the location of size of main memory to be affected by the transfers.

Table 7.2 Intel 8237A Registers

Bit	Command	Status	Mode	Single Mask	All Mask
D0	Memory-to-memory E/D	Channel 0 has reached TC	Channel select	Select channel mask bit	Clear/set channel 0 mask bit
D1	Channel 0 address hold E/D	Channel 1 has reached TC			Clear/set channel 1 mask bit
D2	Controller E/D	Channel 2 has reached TC	Verify/write/read transfer	Clear/set mask bit	Clear/set channel 2 mask bit
D3	Normal/compressed timing	Channel 3 has reached TC		Not used	Clear/set channel 3 mask bit
D4	Fixed/rotating priority	Channel 0 request	Auto-initialization E/D		Not used
D5	Late/extended write selection	Channel 0 request	Address increment/decrement select		
D6	DREQ sense active high/low	Channel 0 request			
D7	DACK sense active high/low	Channel 0 request	Demand/single/block/cascade mode select		

E/D = enable/disable
TC = terminal count

7.6 DIRECT CACHE ACCESS

DMA has proved an effective means of enhancing performance of I/O with peripheral devices and network I/O traffic. However, for the dramatic increases in data rates for network I/O, DMA is not able to scale to meet the increased demand. This demand is coming primarily from the widespread deployment of 10-Gbps and 100-Gbps Ethernet switches to handle massive amounts of data transfer to and from database servers and other high-performance systems [STAL14a]. A secondary but increasingly important source of traffic comes from Wi-Fi in the gigabit range. Network Wi-Fi devices that handle 3.2 Gbps and 6.76 Gbps are becoming widely available and producing demand on enterprise systems [STAL14b].

In this section, we will show how enabling the I/O function to have direct access to the cache can enhance performance, a technique known as **direct cache access (DCA)**. Throughout this section, we are concerned only with the cache that is closest to main memory, referred to as the **last-level cache**. In some systems, this will be an L2 cache, in others an L3 cache.

To begin, we describe the way in which contemporary multicore systems use on-chip shared cache to enhance DMA performance. This approach involves enabling the DMA function to have direct access to the last-level cache. Next we examine cache-related performance issues that manifest when high-speed network traffic is processed. From there, we look at several different strategies for DCA that are designed to enhance network protocol processing performance. Finally, this section describes a DCA approach implemented by Intel, referred to as Direct Data I/O.

DMA Using Shared Last-Level Cache

As was discussed in Chapter 1 (see Figure 1.2), contemporary multicore systems include both cache dedicated to each core and an additional level of shared cache, either L2 or L3. With the increasing size of available last-level cache, system designers have enhanced the DMA function so that the DMA controller has access to the shared cache in a manner similar to the cores. To clarify the interaction of DMA and cache, it will be useful to first describe a specific system architecture. For this purpose, the following is an overview of the Intel Xeon system.

XEON MULTICORE PROCESSOR Intel Xeon is Intel's high-end, high-performance processor family, used in servers, high-performance workstations, and supercomputers. Many of the members of the Xeon family use a ring interconnect system, as illustrated for the Xeon E5-2600/4600 in Figure 7.16.

The E5-2600/4600 can be configured with up to eight cores on a single chip. Each core has dedicated L1 and L2 caches. There is a shared L3 cache of up to 20 MB. The L3 cache is divided into slices, one associated with each core although each core can address the entire cache. Further, each slice has its own cache pipeline, so that requests can be sent in parallel to the slices.

The bidirectional high-speed ring interconnect links cores, last-level cache, PCIe, and integrated memory controller (IMC).

In essence, the ring operates as follows:

1. Each component that attaches to the bidirectional ring (QPI, PCIe, L3 cache, L2 cache) is considered a ring agent, and implements ring agent logic.
2. The ring agents cooperate via a distributed protocol to request and allocate access to the ring, in the form of time slots.
3. When an agent has data to send, it chooses the ring direction that results in the shortest path to the destination and transmits when a scheduling slot is available.

The ring architecture provides good performance and scales well for multiple cores, up to a point. For systems with a greater number of cores, multiple rings are used, with each ring supporting some of the cores.

DMA USE OF THE CACHE In traditional DMA operation, data are exchanged between main memory and an I/O device by means of the system interconnection structure, such as a bus, ring, or QPI point-to-point matrix. So, for example, if the Xeon E5-2600/4600 used a traditional DMA technique, output would proceed as follows. An I/O driver running on a core would send an I/O command to the I/O controller (labeled PCIe in Figure 7.16) with the location and size of the buffer in main memory containing the data to be transferred. The I/O controller issues a read request that is routed to the memory controller hub (MCH), which accesses the data on DDR3 memory and puts it on the system ring for delivery to the I/O controller. The L3 cache is not involved in this transaction and one or more off-chip memory reads are required. Similarly, for input, data arrive from the I/O controller and is delivered over the system ring to the MCH and written out to main memory. The MCH must also invalidate any L3 cache lines corresponding to the updated memory locations. In this case, one or more off-chip memory writes are required. Further, if an application wants to access the new data, a main memory read is required.

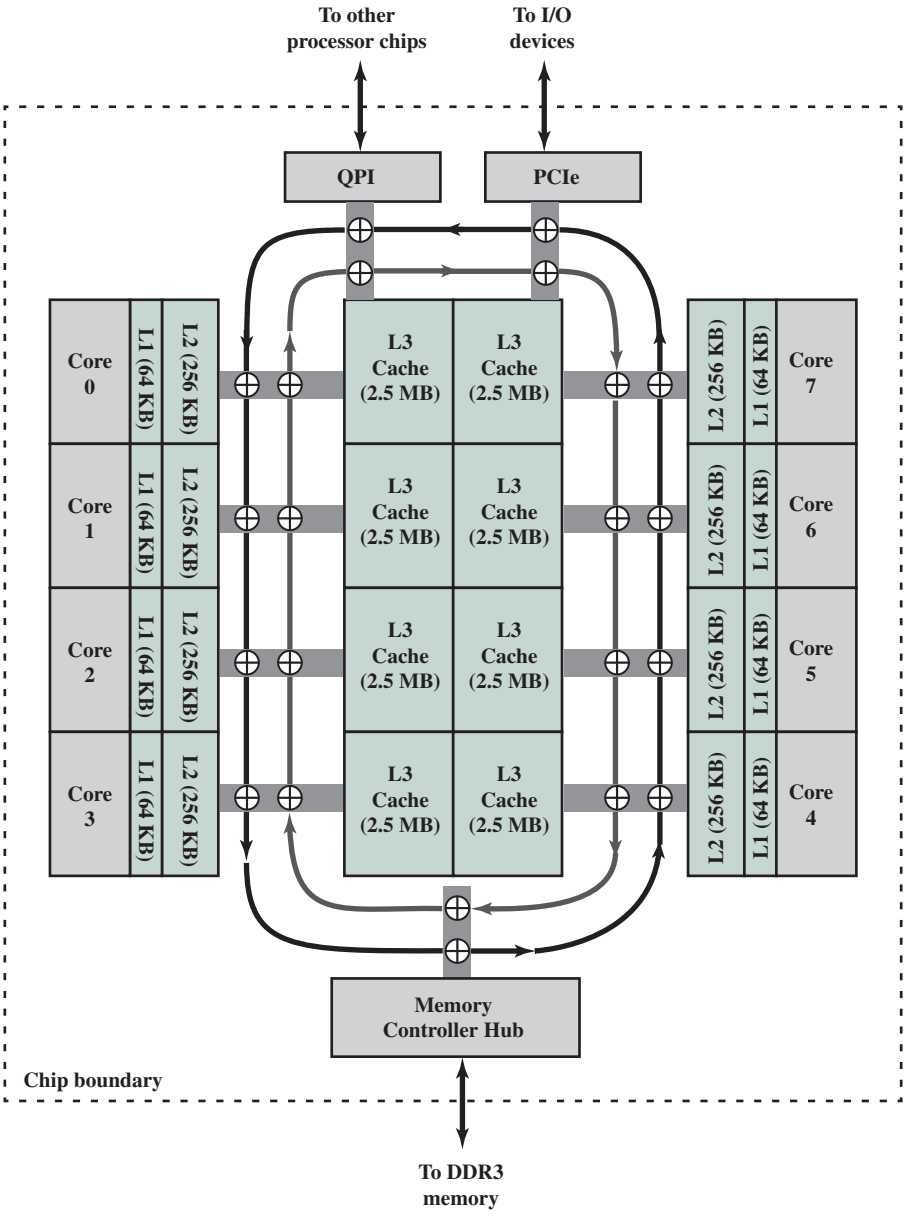


Figure 7.16 Xeon E5-2600/4600 Chip Architecture

With the availability of large amounts of last-level cache, a more efficient technique is possible, and is used by the Xeon E5-2600/4600. For output, when the I/O controller issues a read request, the MCH first checks to see if the data are in the L3 cache. This is likely to be the case, if an application has recently written data into the memory block to be output. In that case, the MCH directs data from the L3 cache to the I/O controller; no main memory accesses are needed. However, it also causes the data to be evicted from cache, that is, the act of reading by an I/O device

causes data to be evicted. Thus, the I/O operation proceeds efficiently because it does not require main memory access. But, if an application does need that data in the future, it must be read back into the L3 cache from main memory. The input operation on the Xeon E5-2600/4600 operates as described in the previous paragraph; the L3 cache is not involved. Thus, the performance improvement involves only output operations.

A final point. Although the output transfer is directly from cache to the I/O controller, the term *direct cache access* is not used for this feature. Rather, the term is reserved for the I/O protocol application, as described in the remainder of this section.

Cache-Related Performance Issues

Network traffic is transmitted in the form of a sequence of protocol blocks, called packets or protocol data units. The lowest, or link, level protocol is typically Ethernet, so that each arriving and departing block of data consists of an Ethernet packet containing as payload the higher-level protocol packet. The higher-level protocols are usually the Internet Protocol (IP), operating on top of Ethernet, and the Transmission Control Protocol (TCP), operating on top of IP. Accordingly, the Ethernet payload consists of a block of data with a TCP header and an IP header. For outgoing data, Ethernet packets are formed in a peripheral component, such as an I/O controller or network interface controller (NIC). Similarly, for incoming traffic, the I/O controller strips off the Ethernet information and delivers the TCP/IP packet to the host CPU.

For both outgoing and incoming traffic, the core, main memory, and cache are all involved. In a DMA scheme, when an application wishes to transmit data, it places that data in an application-assigned buffer in main memory. The core transfers this to a system buffer in main memory and creates the necessary TCP and IP headers, which are also buffered in system memory. The packet is then picked up via DMA for transfer via the NIC. This activity engages not only main memory but also the cache. For incoming traffic, similar transfers between system and application buffers are required.

When large volumes of protocol traffic are processed, two factors in this scenario degrade performance. First, the core consumes valuable clock cycles in copying data between system and application buffers. Second, because memory speeds have not kept up with CPU speeds, the core loses time waiting on memory reads and writes. In this traditional way of processing protocol traffic, the cache does not help because the data and protocol headers are constantly changing and thus the cache must constantly be updated.

To clarify the performance issue and to explain the benefit of DCA as a way of improving performance, let us look at the processing of protocol traffic in more detail for incoming traffic. In general terms, the following steps occur:

1. **Packet arrives:** The NIC receives an incoming Ethernet packet. The NIC processes and strips off the Ethernet control information. This includes doing an error detection calculation. The remaining TCP/IP packet is then transferred to the system's DMA module, which generally is part of the NIC. The NIC also creates a packet descriptor with information about the packet, such as its buffer location in memory.

2. **DMA:** The DMA module transfers data, including the packet descriptor, to main memory. It must also invalidate the corresponding cache lines, if any.
3. **NIC interrupts host:** After a number of packets have been transferred, the NIC issues an interrupt to the host processor.
4. **Retrieve descriptors and headers:** The core processes the interrupt, invoking an interrupt handling procedure, which reads the descriptor and header of the received packets.
5. **Cache miss occurs:** Because this is new data coming in, the cache lines corresponding to the system buffer containing the new data are invalidated. Thus, the core must stall to read the data from main memory into cache, and then to core registers.
6. **Header is processed:** The protocol software executes on the core to analyze the contents of the TCP and IP headers. This will likely include accessing a transport control block (TCB), which contains context information related to TCP. The TCB access may or may not trigger a cache miss, necessitating a main memory access.
7. **Payload transferred:** The data portion of the packet is transferred from the system buffer to the appropriate application buffer.

A similar sequence of steps occurs for outgoing packet traffic, but there are some differences that affect how the cache is managed. For outgoing traffic, the following steps occur:

1. **Packet transfer requested:** When an application has a block of data to transfer to a remote system, it places the data in an application buffer and alerts the OS with some type of system call.
2. **Packet created:** The OS invokes a TCP/IP process to create the TCP/IP packet for transmission. The TCP/IP process accesses the TCB (which may involve a cache miss) and creates the appropriate headers. It also reads the data from the application buffer, and then places the completed packet (headers plus data) in a system buffer. Note that the data that is written into the system buffer also exists in the cache. The TCP/IP process also creates a packet descriptor that is placed in memory shared with the DMA module.
3. **Output operation invoked:** This uses a device driver program to signal the DMA module that output is ready for the NIC.
4. **DMA transfer:** The DMA module reads the packet descriptor, then a DMA transfer is performed from main memory or the last-level cache to the NIC. Note that DMA transfers invalidate the cache line in cache even in the case of a read (by the DMA module). If the line is modified, this causes a write back. The core does not do the invalidates. The invalidates happen when the DMA module reads the data.
5. **NIC signals completion:** After the transfer is complete, the NIC signals the driver on the core that originated the send signal.
6. **Driver frees buffer:** Once the driver receives the completion notice, it frees up the buffer space for reuse. The core must also invalidate the cache lines containing the buffer data.

As can be seen, network I/O involves a number of accesses to cache and main memory and the movement of data between an application buffer and a system buffer. The heavy involvement of main memory becomes a bottleneck, as both core and network performance outstrip gains in memory access times.

Direct Cache Access Strategies

Several strategies have been proposed for making more efficient use of caches for network I/O, with the general term *direct cache access* applied to all of these strategies.

The simplest strategy is one that was implemented as a prototype on a number of Intel Xeon processors between 2006 and 2010 [KUMA07, INTE08]. This form of DCA applies only to incoming network traffic. The DCA function in the memory controller sends a prefetch hint to the core as soon as the data are available in system memory. This enables the core to prefetch the data packet from the system buffer, thus avoiding cache misses and the associated waste of core cycles.

While this simple form of DCA does provide some improvement, much more substantial gains can be realized by avoiding the system buffer in main memory altogether. For the specific function of protocol processing, note that the packet and packet descriptor information are accessed only once in the system buffer by the core. For incoming packets, the core reads the data from the buffer and transfers the packet payload to an application buffer. It has no need to access that data in the system buffer again. Similarly, for outgoing packets, once the core has placed the data in the system buffer, it has no need to access that data again. Suppose, therefore, that the I/O system were equipped not only with the capability of directly accessing main memory, but also of accessing the cache, both for input and output operations. Then it would be possible to use the last-level cache instead of the main memory to buffer packets and descriptors of incoming and outgoing packets.

This last approach, which is a true DCA, was proposed in [HUGG05]. It has also been described as **cache injection** [LEON06]. A version of this more complete form of DCA is implemented in Intel's Xeon processor line, referred to as **Direct Data I/O** [INTE12].

Direct Data I/O

Intel Direct Data I/O (DDIO) is implemented on all of the Xeon E5 family of processors. Its operation is best explained with a side-by-side comparison of transfers with and without DDIO.

PACKET INPUT First, we look at the case of a packet arriving at the NIC from the network. Figure 7.17a shows the steps involved for a DMA operation. The NIC initiates a memory write (1). Then the NIC invalidates the cache lines corresponding to the system buffer (2). Next, the DMA operation is performed, depositing the packet directly into main memory (3). Finally, after the appropriate core receives a DMA interrupt signal, the core can read the packet data from memory through the cache (4).

Before discussing the processing of an incoming packet using DDIO, we need to summarize the discussion of cache write policy from Chapter 4, and introduce a new technique. For the following discussion, there are issues relating to cache coherency that arise in a multiprocessor or multicore environment. These are discussed

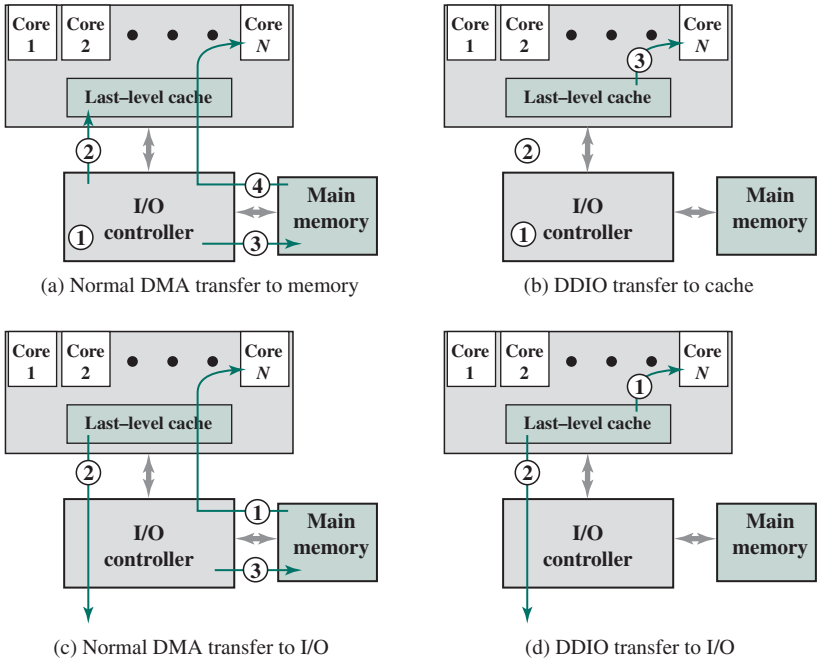


Figure 7.17 Comparison of DMA and DDIO

in Chapter 17 but the details need not concern us here. Recall that there are two techniques for dealing with an update to a cache line:

- **Write through:** All write operations are made to main memory as well as to the cache, ensuring that main memory is always valid. Any other core–cache module can monitor traffic to main memory to maintain consistency within its own local cache.
- **Write back:** Updates are made only in the cache. When an update occurs, a dirty bit associated with the line is set. Then, when a block is replaced, it is written back to main memory if and only if the dirty bit is set.

DDIO uses the write-back strategy in the L3 cache.

A cache write operation may encounter a cache miss, which is dealt with by one of two strategies:

- **Write allocate:** The required line is loaded into the cache from main memory. Then, the line in the cache is updated by the write operation. This scheme is typically used with the write-back method.
- **Non-write allocate:** The block is modified directly in main memory. No change is made to the cache. This scheme is typically used with the write-through method.

With the above in mind, we can describe the DDIO strategy for inbound transfers initiated by the NIC.

1. If there is a cache hit, the cache line is updated, but not main memory; this is simply the write-back strategy for a cache hit. The Intel literature refers to this as **write update**.

2. If there is a cache miss, the write operation occurs to a line in the cache that will not be written back to main memory. Subsequent writes update the cache line, again with no reference to main memory or no future action that writes this data to main memory. The Intel documentation [INTE12] refers to this as *write allocate*, which unfortunately is not the same meaning for the term in the general cache literature.

The DDIO strategy is effective for a network protocol application because the incoming data need not be retained for future use. The protocol application is going to write the data to an application buffer, and there is no need to temporarily store it in a system buffer.

Figure 7.17b shows the operation for DDIO input. The NIC initiates a memory write (1). Then the NIC invalidates the cache lines corresponding to the system buffer and deposits the incoming data in the cache (2). Finally, after the appropriate core receives a DCA interrupt signal, the core can read the packet data from the cache (3).

PACKET OUTPUT Figure 7.17c shows the steps involved for a DMA operation for outbound packet transmission. The TCP/IP protocol handler executing on the core reads data in from an application buffer and writes it out to a system buffer. These data access operations result in cache misses and cause data to be read from memory and into the L3 cache (1). When the NIC receives notification for starting a transmit operation, it reads the data from the L3 cache and transmits it (2). The cache access by the NIC causes the data to be evicted from the cache and written back to main memory (3).

Figure 7.17d shows the steps involved for a DDIO operation for packet transmission. The TCP/IP protocol handler creates the packet to be transmitted and stores it in allocated space in the L3 cache (1), but not in main memory (2). The read operation initiated by the NIC is satisfied by data from the cache, without causing evictions to main memory.

It should be clear from these side-by-side comparisons that DDIO is more efficient than DMA for both incoming and outgoing packets and is therefore better able to keep up with the high packet traffic rate.

7.7 I/O CHANNELS AND PROCESSORS

The Evolution of the I/O Function

As computer systems have evolved, there has been a pattern of increasing complexity and sophistication of individual components. Nowhere is this more evident than in the I/O function. We have already seen part of that evolution. The evolutionary steps can be summarized as follows:

1. The CPU directly controls a peripheral device. This is seen in simple microprocessor-controlled devices.
2. A controller or I/O module is added. The CPU uses programmed I/O without interrupts. With this step, the CPU becomes somewhat divorced from the specific details of external device interfaces.
3. The same configuration as in step 2 is used, but now interrupts are employed. The CPU need not spend time waiting for an I/O operation to be performed, thus increasing efficiency.

4. The I/O module is given direct access to memory via DMA. It can now move a block of data to or from memory without involving the CPU, except at the beginning and end of the transfer.
5. The I/O module is enhanced to become a processor in its own right, with a specialized instruction set tailored for I/O. The CPU directs the I/O processor to execute an I/O program in memory. The I/O processor fetches and executes these instructions without CPU intervention. This allows the CPU to specify a sequence of I/O activities and to be interrupted only when the entire sequence has been performed.
6. The I/O module has a local memory of its own and is, in fact, a computer in its own right. With this architecture, a large set of I/O devices can be controlled, with minimal CPU involvement. A common use for such an architecture has been to control communication with interactive terminals. The I/O processor takes care of most of the tasks involved in controlling the terminals.

As one proceeds along this evolutionary path, more and more of the I/O function is performed without CPU involvement. The CPU is increasingly relieved of I/O-related tasks, improving performance. With the last two steps (5–6), a major change occurs with the introduction of the concept of an I/O module capable of executing a program. For step 5, the I/O module is often referred to as an *I/O channel*. For step 6, the term *I/O processor* is often used. However, both terms are on occasion applied to both situations. In what follows, we will use the term *I/O channel*.

Characteristics of I/O Channels

The I/O channel represents an extension of the DMA concept. An I/O channel has the ability to execute I/O instructions, which gives it complete control over I/O operations. In a computer system with such devices, the CPU does not execute I/O instructions. Such instructions are stored in main memory to be executed by a special-purpose processor in the I/O channel itself. Thus, the CPU initiates an I/O transfer by instructing the I/O channel to execute a program in memory. The program will specify the device or devices, the area or areas of memory for storage, priority, and actions to be taken for certain error conditions. The I/O channel follows these instructions and controls the data transfer.

Two types of I/O channels are common, as illustrated in Figure 7.18. A *selector channel* controls multiple high-speed devices and, at any one time, is dedicated to the transfer of data with one of those devices. Thus, the I/O channel selects one device and effects the data transfer. Each device, or a small set of devices, is handled by a *controller,* or I/O module, that is much like the I/O modules we have been discussing. Thus, the I/O channel serves in place of the CPU in controlling these I/O controllers. A *multiplexor channel* can handle I/O with multiple devices at the same time. For low-speed devices, a *byte multiplexor* accepts or transmits characters as fast as possible to multiple devices. For example, the resultant character stream from three devices with different rates and individual streams $A_1A_2A_3A_4 \ldots$, $B_1B_2B_3B_4 \ldots$, and $C_1C_2C_3C_4 \ldots$ might be $A_1B_1C_1A_2C_2A_3B_2C_3A_4$, and so on. For high-speed devices, a *block multiplexor* interleaves blocks of data from several devices.

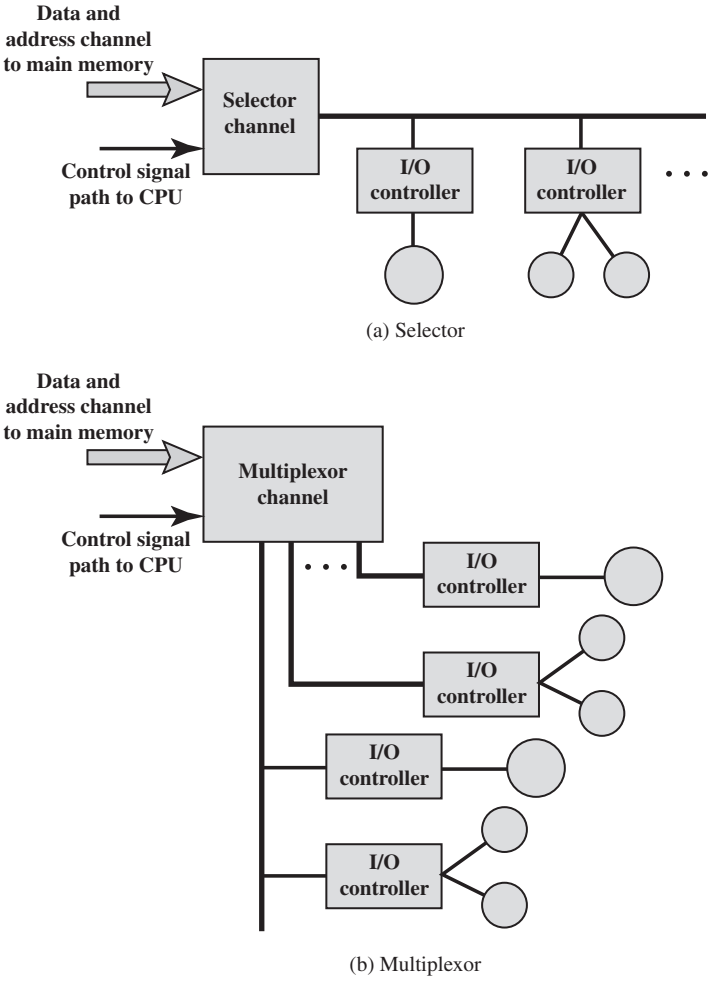


Figure 7.18 I/O Channel Architecture

7.8 EXTERNAL INTERCONNECTION STANDARDS

In this section, we provide a brief overview of the most widely used external interface standards to support I/O. Two of these, Thunderbolt and InfiniBand, are examined in detail in Appendix J.

Universal Serial Bus (USB)

USB is widely used for peripheral connections. It is the default interface for slower-speed devices, such as keyboard and pointing devices, but is also commonly used for high-speed I/O, including printers, disk drives, and network adapters.

USB has gone through multiple generations. The first version, USB 1.0, defined a *Low Speed* data rate of 1.5 Mbps and a *Full Speed* rate of 12 Mbps. USB 2.0 provides a data rate of 480 Mbps. USB 3.0 includes a new, higher speed bus

called *SuperSpeed* in parallel with the USB 2.0 bus. The signaling speed of SuperSpeed is 5 Gbps, but due to signaling overhead, the usable data rate is up to 4 Gbps. The most recent specification is USB 3.1, which includes a faster transfer mode called *SuperSpeed+*. This transfer mode achieves a signaling rate of 10 Gbps and a theoretical usable data rate of 9.7 Gbps.

A USB system is controlled by a root host controller, which attaches to devices to create a local network with a hierarchical tree topology.

FireWire Serial Bus

FireWire was developed as an alternative to the small computer system interface (SCSI) to be used on smaller systems, such as personal computers, workstations, and servers. The objective was to meet the increasing demands for high I/O rates on these systems, while avoiding the bulky and expensive I/O channel technologies developed for mainframe and supercomputer systems. The result is the IEEE standard 1394, for a High Performance Serial Bus, commonly known as FireWire.

FireWire uses a daisy-chain configuration, with up to 63 devices connected off a single port. Moreover, up to 1022 FireWire buses can be interconnected using bridges, enabling a system to support as many peripherals as required.

FireWire provides for what is known as hot plugging, which makes it possible to connect and disconnect peripherals without having to power the computer system down or reconfigure the system. Also, FireWire provides for automatic configuration; it is not necessary manually to set device IDs or to be concerned with the relative position of devices. With FireWire, there are no terminations, and the system automatically performs a configuration function to assign addresses. A FireWire bus need not be a strict daisy chain. Rather, a tree-structured configuration is possible.

An important feature of the FireWire standard is that it specifies a set of three layers of protocols to standardize the way in which the host system interacts with the peripheral devices over the serial bus. The physical layer defines the transmission media that are permissible under FireWire and the electrical and signaling characteristics of each. Data rates from 25 Mbps to 3.2 Gbps are defined. The link layer describes the transmission of data in the packets. The transaction layer defines a request–response protocol that hides the lower-layer details of FireWire from applications.

Small Computer System Interface (SCSI)

SCSI is a once common standard for connecting peripheral devices (disks, modems, printers, etc.) to small and medium-sized computers. Although SCSI has evolved to higher data rates, it has lost popularity to such competitors as USB and FireWire in smaller systems. However, high-speed versions of SCSI remain popular for mass memory support on enterprise systems. For example, the IBM zEnterprise EC12 and other IBM mainframes offer support for SCSI, and a number of Seagate hard drive systems use SCSI.

The physical organization of SCSI is a shared bus, which can support up to 16 or 32 devices, depending on the generation of the standard. The bus provides for parallel transmission rather than serial, with a bus width of 16 bits on earlier generations and 32 bits on later generations. Speeds range from 5 Mbps on the original SCSI-1 specification to 160 Mbps on SCSI-3 U3.

Thunderbolt

The most recent, and one of fastest, peripheral connection technology to become available for general-purpose use is Thunderbolt, developed by Intel with collaboration from Apple. One Thunderbolt cable can manage the work previously required of multiple cables. The technology combines data, video, audio, and power into a single high-speed connection for peripherals such as hard drives, RAID (Redundant Array of Independent Disks) arrays, video-capture boxes, and network interfaces. It provides up to 10 Gbps throughput in each direction and up to 10 watts of power to connected peripherals.

Thunderbolt is described in detail in Appendix J.

InfiniBand

InfiniBand is an I/O specification aimed at the high-end server market. The first version of the specification was released in early 2001 and has attracted numerous vendors. For example, IBM zEnterprise series of mainframes has relied heavily on InfiniBand for a number of years. The standard describes an architecture and specifications for data flow among processors and intelligent I/O devices. InfiniBand has become a popular interface for storage area networking and other large storage configurations. In essence, InfiniBand enables servers, remote storage, and other network devices to be attached in a central fabric of switches and links. The switch-based architecture can connect up to 64,000 servers, storage systems, and networking devices.

Infiniband is described in detail in Appendix J.

PCI Express

PCI Express is a high-speed bus system for connecting peripherals of a wide variety of types and speeds. Chapter 3 discusses PCI Express in detail.

SATA

Serial ATA (Serial Advanced Technology Attachment) is an interface for disk storage systems. It provides data rates of up to 6 Gbps, with a maximum per device of 300 Mbps. SATA is widely used in desktop computers, and in industrial and embedded applications.

Ethernet

Ethernet is the predominant wired networking technology, used in homes, offices, data centers, enterprises, and wide-area networks. As Ethernet has evolved to support data rates up to 100 Gbps and distances from a few meters to tens of km, it has become essential for supporting personal computers, workstations, servers, and massive data storage devices in organizations large and small.

Ethernet began as an experimental bus-based 3-Mbps system. With a bus system, all of the attached devices, such as PCs, connect to a common coaxial cable, much like residential cable TV systems. The first commercially-available Ethernet, and the first version of IEEE 802.3, were bus-based systems operating at 10 Mbps. As technology has advanced, Ethernet has moved from bus-based to switch-based, and the data rate has periodically increased by an order of magnitude. With

switch-based systems, there is a central switch, with all of the devices connected directly to the switch. Currently, Ethernet systems are available at speeds up to 100 Gbps. Here is a brief chronology.

- 1983: 10 Mbps (megabit per second, million bits per second)
- 1995: 100 Mbps
- 1998: 1 Gbps (gigabit per second, billion bits per second)
- 2003: 10 Gbps
- 2010: 40 Gbps and 100 Gbps

Wi-Fi

Wi-Fi is the predominant wireless Internet access technology, used in homes, offices, and public spaces. Wi-Fi in the home now connects computers, tablets, smart phones, and a host of electronic devices, such as video cameras, TVs, and thermostats. Wi-Fi in the enterprise has become an essential means of enhancing worker productivity and network effectiveness. And public Wi-Fi hotspots have expanded dramatically to provide free Internet access in most public places.

As the technology of antennas, wireless transmission techniques, and wireless protocol design has evolved, the IEEE 802.11 committee has been able to introduce standards for new versions of Wi-Fi at ever-higher speeds. Once the standard is issued, industry quickly develops the products. Here is a brief chronology, starting with the original standard, which was simply called IEEE 802.11, and showing the maximum data rate for each version:

- 802.11 (1997): 2 Mbps (megabit per second, million bits per second)
- 802.11a (1999): 54 Mbps
- 802.11b (1999): 11 Mbps
- 802.11n (1999): 600 Mbps
- 802.11g (2003): 54 Mbps
- 802.11ad (2012): 6.76 Gbps (billion bits per second)
- 802.11ac (2014): 3.2 Gbps

7.9 IBM zENTERPRISE EC12 I/O STRUCTURE

The zEnterprise EC12 is IBM's latest mainframe computer offering (at the time of this writing). The system is based on the use of the zEC12 processor chip, which is a 5.5-GHz multicore chip with six cores. The zEC12 architecture can have a maximum of 101 processor chips for a total of 606 cores. In this section, we look at the I/O structure of the zEnterprise EC12.

Channel Structure

The zEnterprise EC12 has a dedicated I/O subsystem that manages all I/O operations, completely off-loading this processing and memory burden from the main

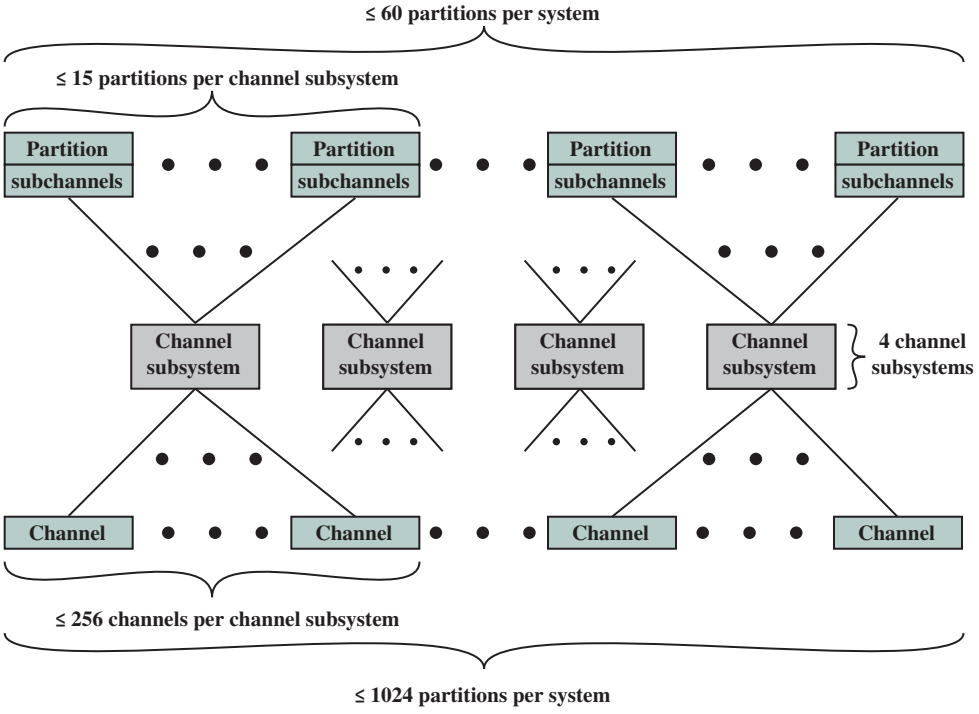


Figure 7.19 IBM zEC12 I/O Channel Subsystem Structure

processors. Figure 7. 21 shows the logical structure of the I/O subsystem. Of the 96 core processors, up to 4 of these can be dedicated for I/O use, creating 4 **channel subsystems (CSS)**. Each CSS is made up of the following elements:

- **System assist processor (SAP):** The SAP is a core processor configured for I/O operation. Its role is to offload I/O operations and manage channels and the I/O operations queues. It relieves the other processors of all I/O tasks, allowing them to be dedicated to application logic.
- **Hardware system area (HSA):** The HSA is a reserved part of the system memory containing the I/O configuration. It is used by SAPs. A fixed amount of 32 GB is reserved, which is not part of the customer-purchased memory. This provides for greater configuration flexibility and higher availability by eliminating planned and preplanned outages.
- **Logical partitions:** A logical partition is a form of virtual machine, which is in essence, a logical processor defined at the operating system level.[3] Each CSS supports up to 16 logical partitions.

[3]A virtual machine is an instance of an operating system along with one or more applications running in an isolated memory partition within the computer. It enables different operating systems to run in the same computer at the same time as well as prevents applications from interfering with each other. See [STAL12] for a discussion of virtual machines.

- **Subchannels:** A subchannel appears to a program as a logical device and contains the information required to perform an I/O operation. One subchannel exists for each I/O device addressable by the CSS. A subchannel is used by the channel subsystem code running on a partition to pass an I/O request to the channel subsystem. A subchannel is assigned for each device defined to the logical partition. Up to 196k subchannels are supported per CSS.
- **Channel path:** A channel path is a single interface between a channel subsystem and one or more control units, via a channel. Commands and data are sent across a channel path to perform I/O requests. Each CSS can have up to 256 channel paths.
- **Channel:** Channels are small processors that communicate with the I/O control units (CUs). They manage the data transfer between memory and the external devices.

This elaborate structure enables the mainframe to manage a massive number of I/O devices and communication links. All I/O processing is offloaded from the application and server processors, enhancing performance. The channel subsystem processors are somewhat general in configuration, enabling them to manage a wide variety of I/O duties and to keep up with evolving requirements. The channel processors are specifically programmed for the I/O control units to which they interface.

I/O System Organization

To explain the I/O system organization, we need to first briefly explain the physical layout of the zEnterprise EC12. Figure 7.20 is a front view of the water-cooled version of the machine (there is also an air-cooled version). The system has the following characteristics:

- Weight: 2430 kg (5358 lbs)
- Width: 1.568 m (5.14 ft)
- Depth: 1.69 m (6.13 ft)
- Height: 2.015 m (6.6 ft)

Not exactly a laptop.

The system consists of two large bays, called frames, that house the various components of the zEnterprise EC12. The right-hand A frame includes two large cages, plus room for cabling and other components. The upper cage is a processor cage, with four slots to house up to four processor books that are fully interconnected. Each book contains a multichip module (MCM), memory cards, and I/O cage connections. Each MCM is a board that houses six multicore chips and two storage control chips.

The lower cage in the A frame is an I/O cage, which contains I/O hardware, including multiplexors and channels. The I/O cage is a fixed unit installed by IBM to the customer specifications at the factory.

The left-hand Z frame contains internal batteries and power supplies and room for one or more support elements, which are used by a system manager for platform management. The Z frame also contains slots for two or more I/O drawers.

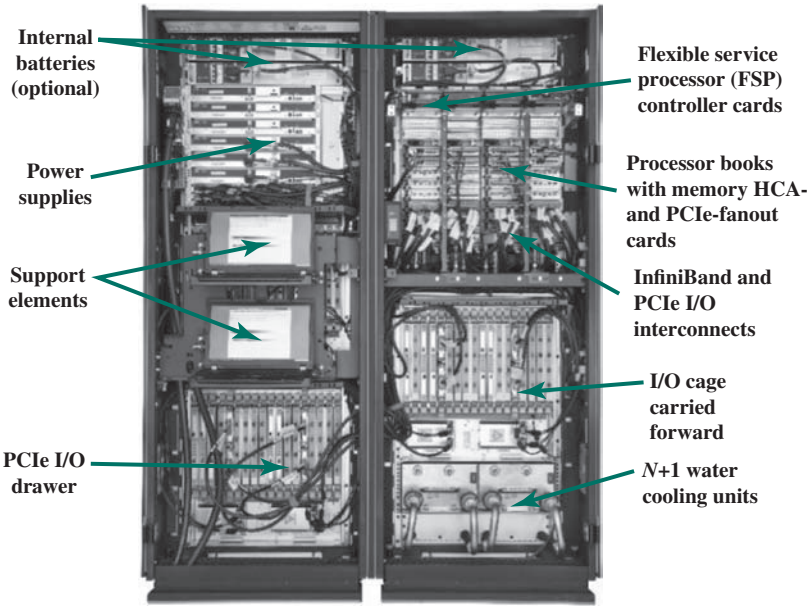

Internal batteries (optional)
Power supplies
Support elements
PCIe I/O drawer
Flexible service processor (FSP) controller cards
Processor books with memory HCA- and PCIe-fanout cards
InfiniBand and PCIe I/O interconnects
I/O cage carried forward
N+1 water cooling units

Figure 7.20 IBM zEC12 I/O Frames–Front View

An I/O drawer contains similar components to an I/O cage. The differences are that the drawer is smaller and easily swapped in and out at the customer site to meet changing requirements.

With this background, we now show a typical configuration of the zEnterprise EC12 I/O system structure (Figure 7.21). Each zEC12 processor book supports two internal (i.e., internal to the A and Z frames) I/O infrastructures: InfiniBand for I/O cages and I/O drawers, and PCI Express (PCIe) for I/O drawers. These channel controllers are referred to as **fanouts**.

The InfiniBand connections from the processor book to the I/O cages and I/O drawers are via a Host Channel Adapter (HCA) fanout, which has InfiniBand links to InfiniBand multiplexors in the I/O cage or drawer. The InfiniBand multiplexors are used to interconnect servers, communications infrastructure equipment, storage, and embedded systems. In addition to using InfiniBand to interconnect systems, all of which use InfiniBand, the InfiniBand multiplexor supports other I/O technologies. ESCON (Enterprise Systems Connection) supports connectivity to disks, tapes, and printer devices using a proprietary fiber-based technology. Ethernet connections provide 1-Gbps and 10-Gbps connections to a variety of devices that support this popular local area network technology. One noteworthy use of Ethernet is to construct large server farms, particularly to interconnect blade servers with each other and with other mainframes.[4]

[4]A blade server is a server architecture that houses multiple server modules (blades) in a single chassis. It is widely used in data centers to save space and improve system management. Either self-standing or rack mounted, the chassis provides the power supply, and each blade has its own CPU, memory, and hard disk.

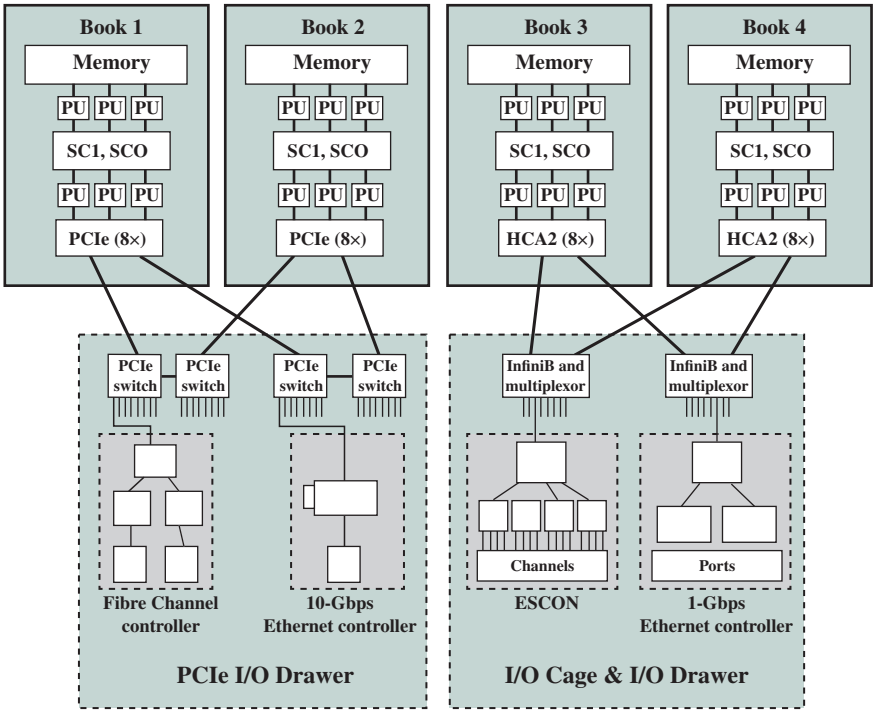


Figure 7.21 IBM zEC12 I/O System Structure

The PCIe connections from the processor book to the I/O drawers are via a PCIe fanout to PCIe switches. The PCIe switches can connect to a number of I/O device controllers. Typical examples for zEnterprise EC12 are 1-Gbps and 10-Gbps Ethernet and Fiber Channel.

Each book contains a combination of up to 8 InfiniBand HCA and PCIe fanouts. Each fanout supports up to 32 connections, for a total maximum of 256 connections per processor book, each connection controlled by a channel processor.

7.10 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

cache injection	I/O command	peripheral device
cycle stealing	I/O module	programmed I/O
direct cache access (DCA)	I/O processor	selector channel
Direct Data I/O	isolated I/O	serial I/O
direct memory access (DMA)	last-level cache	Thunderbolt
InfiniBand	memory-mapped I/O	write allocate
interrupt	multiplexor channel	write back
interrupt-driven I/O	non-write allocate	write through
I/O channel	parallel I/O	write update

Review Questions

7.1 List three broad classifications of external, or peripheral, devices.

7.2 What is the International Reference Alphabet?

7.3 What are the major functions of an I/O module?

7.4 List and briefly define three techniques for performing I/O.

7.5 What is the difference between memory-mapped I/O and isolated I/O?

7.6 When a device interrupt occurs, how does the processor determine which device issued the interrupt?

7.7 When a DMA module takes control of a bus, and while it retains control of the bus, what does the processor do?

Problems

7.1 On a typical microprocessor, a distinct I/O address is used to refer to the I/O data registers and a distinct address for the control and status registers in an I/O controller for a given device. Such registers are referred to as **ports**. In the Intel 8088, two I/O instruction formats are used. In one format, the 8-bit opcode specifies an I/O operation; this is followed by an 8-bit port address. Other I/O opcodes imply that the port address is in the 16-bit DX register. How many ports can the 8088 address in each I/O addressing mode?

7.2 A similar instruction format is used in the Zilog Z8000 microprocessor family. In this case, there is a direct port addressing capability, in which a 16-bit port address is part of the instruction, and an indirect port addressing capability, in which the instruction references one of the 16-bit general purpose registers, which contains the port address. How many ports can the Z8000 address in each I/O addressing mode?

7.3 The Z8000 also includes a block I/O transfer capability that, unlike DMA, is under the direct control of the processor. The block transfer instructions specify a port address register (Rp), a count register (Rc), and a destination register (Rd). Rd contains the main memory address at which the first byte read from the input port is to be stored. Rc is any of the 16-bit general purpose registers. How large a data block can be transferred?

7.4 Consider a microprocessor that has a block I/O transfer instruction such as that found on the Z8000. Following its first execution, such an instruction takes five clock cycles to re-execute. However, if we employ a nonblocking I/O instruction, it takes a total of 20 clock cycles for fetching and execution. Calculate the increase in speed with the block I/O instruction when transferring blocks of 128 bytes.

7.5 A system is based on an 8-bit microprocessor and has two I/O devices. The I/O controllers for this system use separate control and status registers. Both devices handle data on a 1-byte-at-a-time basis. The first device has two status lines and three control lines. The second device has three status lines and four control lines.

- **a.** How many 8-bit I/O control module registers do we need for status reading and control of each device?
- **b.** What is the total number of needed control module registers given that the first device is an output-only device?
- **c.** How many distinct addresses are needed to control the two devices?

7.6 For programmed I/O, Figure 7.5 indicates that the processor is stuck in a wait loop doing status checking of an I/O device. To increase efficiency, the I/O software could be written so that the processor periodically checks the status of the device. If the device is not ready, the processor can jump to other tasks. After some timed interval, the processor comes back to check status again.

- **a.** Consider the above scheme for outputting data one character at a time to a printer that operates at 10 characters per second (cps). What will happen if its status is scanned every 200 ms?

b. Next consider a keyboard with a single character buffer. On average, characters are entered at a rate of 10 cps. However, the time interval between two consecutive key depressions can be as short as 60 ms. At what frequency should the keyboard be scanned by the I/O program?

7.7 A microprocessor scans the status of an output I/O device every 20 ms. This is accomplished by means of a timer alerting the processor every 20 ms. The interface of the device includes two ports: one for status and one for data output. How long does it take to scan and service the device, given a clocking rate of 8 MHz? Assume for simplicity that all pertinent instruction cycles take 12 clock cycles.

7.8 In Section 7.3, one advantage and one disadvantage of memory-mapped I/O, compared with isolated I/O, were listed. List two more advantages and two more disadvantages.

7.9 A particular system is controlled by an operator through commands entered from a keyboard. The average number of commands entered in an 8-hour interval is 60.

a. Suppose the processor scans the keyboard every 100 ms. How many times will the keyboard be checked in an 8-hour period?

b. By what fraction would the number of processor visits to the keyboard be reduced if interrupt-driven I/O were used?

7.10 Suppose that the 8255A shown in Figure 7.9 is configured as follows: port A as input, port B as output, and all the bits of port C as output. Show the bits of the control register to define this configuration.

7.11 Consider a system employing interrupt-driven I/O for a particular device that transfers data at an average of 8 KB/s on a continuous basis.

a. Assume that interrupt processing takes about 100 μs (i.e., the time to jump to the interrupt service routine (ISR), execute it, and return to the main program). Determine what fraction of processor time is consumed by this I/O device if it interrupts for every byte.

b. Now assume that the device has two 16-byte buffers and interrupts the processor when one of the buffers is full. Naturally, interrupt processing takes longer, because the ISR must transfer 16 bytes. While executing the ISR, the processor takes about 8 μs for the transfer of each byte. Determine what fraction of processor time is consumed by this I/O device in this case.

c. Now assume that the processor is equipped with a block transfer I/O instruction such as that found on the Z8000. This permits the associated ISR to transfer each byte of a block in only 2 μs. Determine what fraction of processor time is consumed by this I/O device in this case.

7.12 In virtually all systems that include DMA modules, DMA to main memory is given higher priority than CPU access to main memory. Why?

7.13 A DMA module is transferring characters to memory using **cycle stealing**, from a device transmitting at 9600 bps. The processor is fetching instructions at the rate of 1 million instructions per second (1 MIPS). By how much will the processor be slowed down due to the DMA activity?

7.14 Consider a system in which bus cycles takes 500 ns. Transfer of bus control in either direction, from processor to I/O device or vice versa, takes 250 ns. One of the I/O devices has a data transfer rate of 50 KB/s and employs DMA. Data are transferred 1 byte at a time.

a. Suppose we employ DMA in a burst mode. That is, the DMA interface gains bus mastership prior to the start of a block transfer and maintains control of the bus until the whole block is transferred. For how long would the device tie up the bus when transferring a block of 128 bytes?

b. Repeat the calculation for cycle-stealing mode.

7.15 Examination of the timing diagram of the 8237A indicates that once a block transfer begins, it takes three bus clock cycles per DMA cycle. During the DMA cycle, the 8237A transfers one byte of information between memory and I/O device.

a. Suppose we clock the 8237A at a rate of 5 MHz. How long does it take to transfer one byte?

b. What would be the maximum attainable data transfer rate?
c. Assume that the memory is not fast enough and we have to insert two wait states per DMA cycle. What will be the actual data transfer rate?

7.16 Assume that in the system of the preceding problem, a memory cycle takes 750 ns. To what value could we reduce the clocking rate of the bus without effect on the attainable data transfer rate?

7.17 A DMA controller serves four receive-only telecommunication links (one per DMA channel) having a speed of 64 Kbps each.
a. Would you operate the controller in burst mode or in cycle-stealing mode?
b. What priority scheme would you employ for service of the DMA channels?

7.18 A 32-bit computer has two selector channels and one multiplexor channel. Each selector channel supports two magnetic disk and two magnetic tape units. The multiplexor channel has two line printers, two card readers, and 10 VDT terminals connected to it. Assume the following transfer rates:

Disk drive	800 Kbytes/s
Magnetic tape drive	200 Kbytes/s
Line printer	6.6 Kbytes/s
Card reader	1.2 Kbytes/s
VDT	1 Kbyte/s

Estimate the maximum aggregate I/O transfer rate in this system.

7.19 A computer consists of a processor and an I/O device D connected to main memory M via a shared bus with a data bus width of one word. The processor can execute a maximum of 10^6 instructions per second. An average instruction requires five machine cycles, three of which use the memory bus. A memory read or write operation uses one machine cycle. Suppose that the processor is continuously executing "background" programs that require 95% of its instruction execution rate but not any I/O instructions. Assume that one processor cycle equals one bus cycle. Now suppose the I/O device is to be used to transfer very large blocks of data between M and D.
a. If programmed I/O is used and each one-word I/O transfer requires the processor to execute two instructions, estimate the maximum I/O data-transfer rate, in words per second, possible through D.
b. Estimate the same rate if DMA is used.

7.20 A data source produces 7-bit IRA characters, to each of which is appended a parity bit. Derive an expression for the maximum effective data rate (rate of IRA data bits) over an R-bps line for the following:
a. Asynchronous transmission, with a 1.5-unit stop bit;
b. Bit-synchronous transmission, with a frame consisting of 48 control bits and 128 information bits;
c. Same as (b), with a 1024-bit information field;
d. Character-synchronous, with nine control characters per frame and 16 information characters;
e. Same as (d), with 128 information characters.

7.21 Two women are on either side of a high fence. One of the women, named Apple-server, has a beautiful apple tree loaded with delicious apples growing on her side of the fence; she is happy to supply apples to the other woman whenever needed. The other woman, named Apple-eater, loves to eat apples but has none. In fact, she must eat her apples at a fixed rate (an apple a day keeps the doctor away). If she eats them faster than that rate, she will get sick. If she eats them slower, she will suffer malnutrition. Neither woman can talk, and so the problem is to get apples from Apple-server to Apple-eater at the correct rate.
a. Assume that there is an alarm clock sitting on top of the fence and that the clock can have multiple alarm settings. How can the clock be used to solve the problem? Draw a timing diagram to illustrate the solution.

b. Now assume that there is no alarm clock. Instead Apple-eater has a flag that she can wave whenever she needs an apple. Suggest a new solution. Would it be helpful for Apple-server also to have a flag? If so, incorporate this into the solution. Discuss the drawbacks of this approach.

c. Now take away the flag and assume the existence of a long piece of string. Suggest a solution that is superior to that of (b) using the string.

7.22 Assume that one 16-bit and two 8-bit microprocessors are to be interfaced to a system bus. The following details are given:

1. All microprocessors have the hardware features necessary for any type of data transfer: programmed I/O, interrupt-driven I/O, and DMA.

2. All microprocessors have a 16-bit address bus.

3. Two memory boards, each of 64-Kbytes capacity, are interfaced with the bus. The designer wishes to use a shared memory that is as large as possible.

4. The system bus supports a maximum of four interrupt lines and one DMA line. Make any other assumptions necessary, and:

a. Give the system bus specifications in terms of number and types of lines.

b. Describe a possible protocol for communicating on the bus (i.e., read-write, interrupt, and DMA sequences).

c. Explain how the aforementioned devices are interfaced to the system bus.

CHAPTER 8

OPERATING SYSTEM SUPPORT

8.1 Operating System Overview
- Operating System Objectives and Functions
- Types of Operating Systems

8.2 Scheduling
- Long-Term Scheduling
- Medium-Term Scheduling
- Short-Term Scheduling

8.3 Memory Management
- Swapping
- Partitioning
- Paging
- Virtual Memory
- Translation Lookaside Buffer
- Segmentation

8.4 Intel x86 Memory Management
- Address Spaces
- Segmentation
- Paging

8.5 ARM Memory Management
- Memory System Organization
- Virtual Memory Address Translation
- Memory-Management Formats
- Access Control

8.6 Key Terms, Review Questions, and Problems

Learning Objectives

After studying this chapter, you should be able to:

- Summarize, at a top level, the key functions of an **operating system (OS)**.
- Discuss the evolution of operating systems for early simple batch systems to modern complex systems.
- Explain the differences among long-, medium-, and short-term scheduling.
- Understand the reason for memory **partitioning** and explain the various techniques that are used.
- Assess the relative advantages of paging and segmentation.
- Define virtual memory.

Although the focus of this text is computer hardware, there is one area of software that needs to be addressed: the computer's OS. The OS is a program that manages the computer's resources, provides services for programmers, and schedules the execution of other programs. Some understanding of operating systems is essential to appreciate the mechanisms by which the CPU controls the computer system. In particular, explanations of the effect of interrupts and of the management of the memory hierarchy are best explained in this context.

The chapter begins with an overview and brief history of operating systems. The bulk of the chapter looks at the two OS functions that are most relevant to the study of computer organization and architecture: scheduling and memory management.

8.1 OPERATING SYSTEM OVERVIEW

Operating System Objectives and Functions

An OS is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware. It can be thought of as having two objectives:

- **Convenience:** An OS makes a computer more convenient to use.
- **Efficiency:** An OS allows the computer system resources to be used in an efficient manner.

Let us examine these two aspects of an OS in turn.

THE OPERATING SYSTEM AS A USER/COMPUTER INTERFACE The hardware and software used in providing applications to a user can be viewed in a layered or hierarchical fashion, as depicted in Figure 8.1. The user of those applications, the end user, generally is not concerned with the computer's architecture. Thus the end user views a computer system in terms of an application. That application can be expressed in a programming language and is developed by an application programmer. To develop an application program as a set of processor instructions

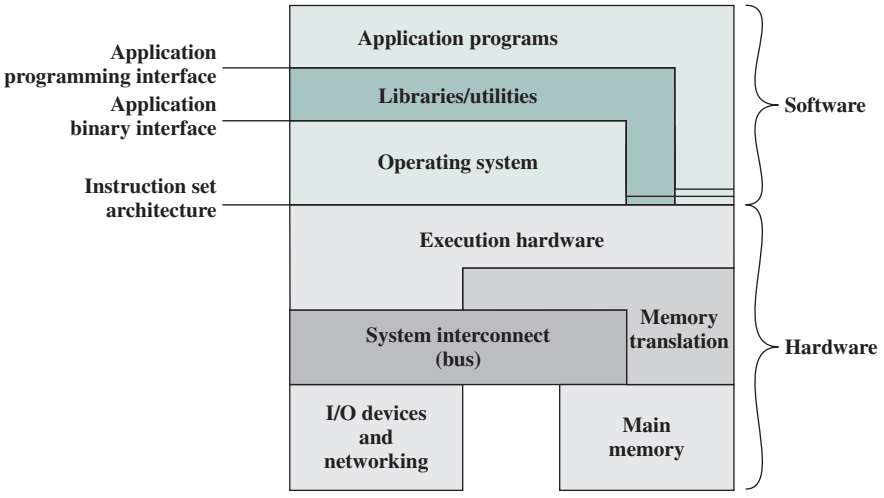


Figure 8.1 Computer Hardware and Software Structure

that is completely responsible for controlling the computer hardware would be an overwhelmingly complex task. To ease this task, a set of system programs is provided. Some of these programs are referred to as **utilities.** These implement frequently used functions that assist in program creation, the management of files, and the control of I/O devices. A programmer makes use of these facilities in developing an application, and the application, while it is running, invokes the utilities to perform certain functions. The most important system program is the OS. The OS masks the details of the hardware from the programmer and provides the programmer with a convenient interface for using the system. It acts as mediator, making it easier for the programmer and for application programs to access and use those facilities and services.

Briefly, the OS typically provides services in the following areas:

- **Program creation:** The OS provides a variety of facilities and services, such as editors and debuggers, to assist the programmer in creating programs. Typically, these services are in the form of **utility** programs that are not actually part of the OS but are accessible through the OS.
- **Program execution:** A number of steps need to be performed to execute a program. Instructions and data must be loaded into main memory, I/O devices and files must be initialized, and other resources must be prepared. The OS handles all of this for the user.
- **Access to I/O devices:** Each I/O device requires its own specific set of instructions or control signals for operation. The OS takes care of the details so that the programmer can think in terms of simple reads and writes.
- **Controlled access to files:** In the case of files, control must include an understanding of not only the nature of the I/O device (disk drive, tape drive) but also the file format on the storage medium. Again, the OS worries about the details. Further, in the case of a system with multiple simultaneous users, the OS can provide protection mechanisms to control access to the files.

- **System access:** In the case of a shared or public system, the OS controls access to the system as a whole and to specific system resources. The access function must provide protection of resources and data from unauthorized users and must resolve conflicts for resource contention.
- **Error detection and response:** A variety of errors can occur while a computer system is running. These include internal and external hardware errors, such as a memory error, or a device failure or malfunction; and various software errors, such as arithmetic overflow, attempt to access forbidden memory location, and inability of the OS to grant the request of an application. In each case, the OS must make the response that clears the error condition with the least impact on running applications. The response may range from ending the program that caused the error, to retrying the operation, to simply reporting the error to the application.
- **Accounting:** A good OS collects usage statistics for various resources and monitors performance parameters such as response time. On any system, this information is useful in anticipating the need for future enhancements and in tuning the system to improve performance. On a multiuser system, the information can be used for billing purposes. Figure 8.1 also indicates three key interfaces in a typical computer system:
  - **Instruction set architecture (ISA):** The ISA defines the repertoire of machine language instructions that a computer can follow. This interface is the boundary between hardware and software. Note that both application programs and utilities may access the ISA directly. For these programs, a subset of the instruction repertoire is available (user ISA). The OS has access to additional machine language instructions that deal with managing system resources (system ISA).
  - **Application binary interface (ABI):** The ABI defines a standard for binary portability across programs. The ABI defines the system call interface to the operating system and the hardware resources and services available in a system through the user ISA.
  - **Application programming interface (API):** The API gives a program access to the hardware resources and services available in a system through the user ISA supplemented with **high-level language (HLL)** library calls. Any system calls are usually performed through libraries. Using an API enables application software to be ported easily, through recompilation, to other systems that support the same API.

THE OPERATING SYSTEM AS RESOURCE MANAGER A computer is a set of resources for the movement, storage, and processing of data and for the control of these functions. The OS is responsible for managing these resources.

Can we say that the OS controls the movement, storage, and processing of data? From one point of view, the answer is yes: By managing the computer's resources, the OS is in control of the computer's basic functions. But this control is exercised in a curious way. Normally, we think of a control mechanism as something external to that which is controlled, or at least as something that is a distinct and separate part of that which is controlled. (For example, a residential heating system

is controlled by a thermostat, which is completely distinct from the heat-generation and heat-distribution apparatus.) This is not the case with the OS, which as a control mechanism is unusual in two respects:

- The OS functions in the same way as ordinary computer software; that is, it is a program executed by the processor.
- The OS frequently relinquishes control and must depend on the processor to allow it to regain control.

Like other computer programs, the OS provides instructions for the processor. The key difference is in the intent of the program. The OS directs the processor in the use of the other system resources and in the timing of its execution of other programs. But in order for the processor to do any of these things, it must cease executing the OS program and execute other programs. Thus, the OS relinquishes control for the processor to do some "useful" work and then resumes control long enough to prepare the processor to do the next piece of work. The mechanisms involved in all this should become clear as the chapter proceeds.

Figure 8.2 suggests the main resources that are managed by the OS. A portion of the OS is in main memory. This includes the **kernel**, or **nucleus**, which contains the most frequently used functions in the OS and, at a given time, other portions of the OS currently in use. The remainder of main memory contains user programs and data. The allocation of this resource (main memory) is controlled jointly by the OS and memory-management hardware in the processor, as we will see. The OS decides when an I/O device can be used by a program in execution, and controls access to and

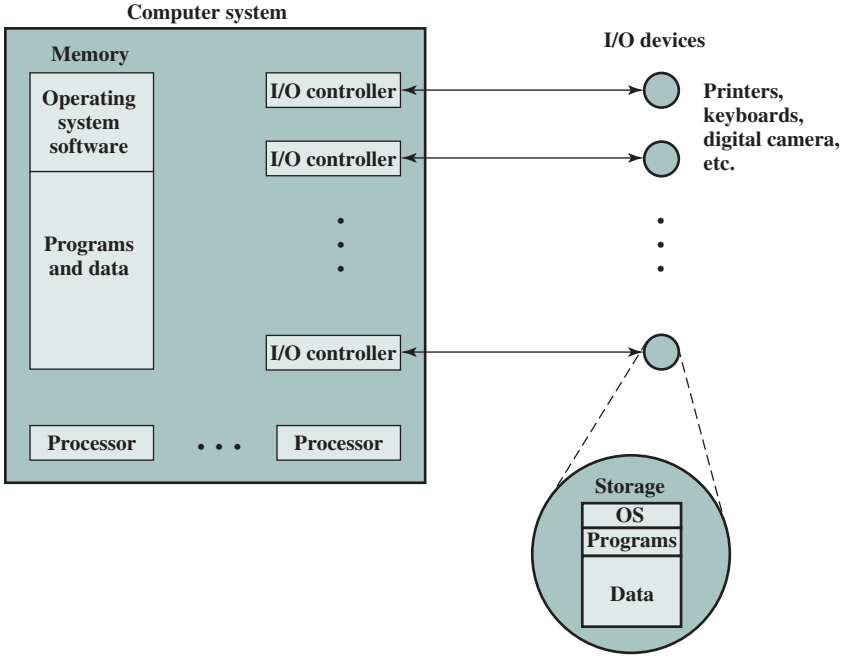


Figure 8.2 The Operating System as Resource Manager

use of files. The processor itself is a resource, and the OS must determine how much processor time is to be devoted to the execution of a particular user program. In the case of a multiple-processor system, this decision must span all of the processors.

Types of Operating Systems

Certain key characteristics serve to differentiate various types of operating systems. The characteristics fall along two independent dimensions. The first dimension specifies whether the system is batch or interactive. In an **interactive** system, the user/programmer interacts directly with the computer, usually through a keyboard/display terminal, to request the execution of a job or to perform a transaction. Furthermore, the user may, depending on the nature of the application, communicate with the computer during the execution of the job. A **batch system** is the opposite of interactive. The user's program is batched together with programs from other users and submitted by a computer operator. After the program is completed, results are printed out for the user. Pure batch systems are rare today, however, it will be useful to the description of contemporary operating systems to briefly examine batch systems.

An independent dimension specifies whether the system employs **multiprogramming** or not. With multiprogramming, the attempt is made to keep the processor as busy as possible, by having it work on more than one program at a time. Several programs are loaded into memory, and the processor switches rapidly among them. The alternative is a **uniprogramming** system that works only one program at a time.

EARLY SYSTEMS With the earliest computers, from the late 1940s to the mid-1950s, the programmer interacted directly with the computer hardware; there was no OS. These processors were run from a console, consisting of display lights, toggle switches, some form of input device, and a printer. Programs in processor code were loaded via the input device (e.g., a card reader). If an error halted the program, the error condition was indicated by the lights. The programmer could proceed to examine registers and main memory to determine the cause of the error. If the program proceeded to a normal completion, the output appeared on the printer.

These early systems presented two main problems:

- **Scheduling:** Most installations used a sign-up sheet to reserve processor time. Typically, a user could sign up for a block of time in multiples of a half hour or so. A user might sign up for an hour and finish in 45 minutes; this would result in wasted computer idle time. On the other hand, the user might run into problems, not finish in the allotted time, and be forced to stop before resolving the problem.
- **Setup time:** A single program, called a **job**, could involve loading the compiler plus the high-level language program (source program) into memory, saving the compiled program (object program), and then loading and linking together the object program and common functions. Each of these steps could involve mounting or dismounting tapes, or setting up card decks. If an error occurred, the hapless user typically had to go back to the beginning of the setup sequence. Thus a considerable amount of time was spent just in setting up the program to run.

This mode of operation could be termed serial processing, reflecting the fact that users have access to the computer in series. Over time, various system software tools were developed to attempt to make serial processing more efficient. These include libraries of common functions, linkers, loaders, debuggers, and I/O driver routines that were available as common software for all users.

SIMPLE BATCH SYSTEMS Early processors were very expensive, and therefore it was important to maximize processor utilization. The wasted time due to scheduling and setup time was unacceptable.

To improve utilization, simple batch operating systems were developed. With such a system, also called a **monitor**, the user no longer has direct access to the processor. Rather, the user submits the job on cards or tape to a computer operator, who *batches* the jobs together sequentially and places the entire batch on an input device, for use by the monitor.

To understand how this scheme works, let us look at it from two points of view: that of the monitor and that of the processor. From the point of view of the monitor, the monitor controls the sequence of events. For this to be so, much of the monitor must always be in main memory and available for execution (Figure 8.3). That portion is referred to as the **resident monitor**. The rest of the monitor consists of utilities and common functions that are loaded as subroutines to the user program at the beginning of any job that requires them. The monitor reads in jobs one at a time from the input device (typically a card reader or magnetic tape drive). As it is read in, the current job is placed in the user program area, and control is passed to this job. When the job is completed, it returns control to the monitor, which immediately reads in the next job. The results of each job are printed out for delivery to the user.

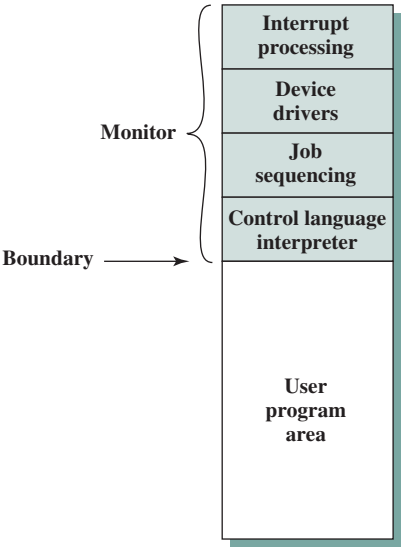


Figure 8.3 Memory Layout for a Resident Monitor

Now consider this sequence from the point of view of the processor. At a certain point in time, the processor is executing instructions from the portion of main memory containing the monitor. These instructions cause the next job to be read in to another portion of main memory. Once a job has been read in, the processor will encounter in the monitor a branch instruction that instructs the processor to continue execution at the start of the user program. The processor will then execute the instruction in the user's program until it encounters an ending or error condition. Either event causes the processor to fetch its next instruction from the monitor program. Thus the phrase "control is passed to a job" simply means that the processor is now fetching and executing instructions in a user program, and "control is returned to the monitor" means that the processor is now fetching and executing instructions from the monitor program.

It should be clear that the monitor handles the scheduling problem. A batch of jobs is queued up, and jobs are executed as rapidly as possible, with no intervening idle time.

How about the job setup time? The monitor handles this as well. With each job, instructions are included in a **job control language (JCL)**. This is a special type of programming language used to provide instructions to the monitor. A simple example is that of a user submitting a program written in FORTRAN plus some data to be used by the program. Each FORTRAN instruction and each item of data is on a separate punched card or a separate record on tape. In addition to FORTRAN and data lines, the job includes job control instructions, which are denoted by the beginning "$". The overall format of the job looks like this:

```
$JOB
$FTN
.              } FORTRAN instructions
.
.
$LOAD
$RUN
.              } Data
.
.
$END
```

To execute this job, the monitor reads the $FTN line and loads the appropriate compiler from its mass storage (usually tape). The compiler translates the user's program into object code, which is stored in memory or mass storage. If it is stored in memory, the operation is referred to as "compile, load, and go." If it is stored on tape, then the $LOAD instruction is required. This instruction is read by the monitor, which regains control after the compile operation. The monitor invokes the loader, which loads the object program into memory in place of the compiler and transfers control to it. In this manner, a large segment of main memory can be shared among different subsystems, although only one such subsystem could be resident and executing at a time.

We see that the monitor, or batch OS, is simply a computer program. It relies on the ability of the processor to fetch instructions from various portions of main

memory in order to seize and relinquish control alternately. Certain other hardware features are also desirable:

- **Memory protection:** While the user program is executing, it must not alter the memory area containing the monitor. If such an attempt is made, the processor hardware should detect an error and transfer control to the monitor. The monitor would then abort the job, print out an error message, and load the next job.
- **Timer:** A timer is used to prevent a single job from monopolizing the system. The timer is set at the beginning of each job. If the timer expires, an interrupt occurs, and control returns to the monitor.
- **Privileged instructions:** Certain instructions are designated privileged and can be executed only by the monitor. If the processor encounters such an instruction while executing a user program, an error interrupt occurs. Among the privileged instructions are I/O instructions, so that the monitor retains control of all I/O devices. This prevents, for example, a user program from accidentally reading job control instructions from the next job. If a user program wishes to perform I/O, it must request that the monitor perform the operation for it. If a privileged instruction is encountered by the processor while it is executing a user program, the processor hardware considers this an error and transfers control to the monitor.
- **Interrupts:** Early computer models did not have this capability. This feature gives the OS more flexibility in relinquishing control to and regaining control from user programs.

Processor time alternates between execution of user programs and execution of the monitor. There have been two sacrifices: Some main memory is now given over to the monitor and some processor time is consumed by the monitor. Both of these are forms of overhead. Even with this overhead, the simple batch system improves utilization of the computer.

MULTIPROGRAMMED BATCH SYSTEMS Even with the automatic job sequencing provided by a simple batch OS, the processor is often idle. The problem is that I/O devices are slow compared to the processor. Figure 8.4 details a representative calculation. The calculation concerns a program that processes a file of records and performs, on average, 100 processor instructions per record. In this example the computer spends over 96% of its time waiting for I/O devices to finish transferring data! Figure 8.5a illustrates this situation. The processor spends a certain amount of

Read one record from file	**15 μs**
Execute 100 instructions	**1 μs**
Write one record to file	**15 μs**
TOTAL	**31 μs**

Percent CPU utilization $= \frac{1}{31} = 0.032 = 3.2\%$

Figure 8.4 System Utilization Example

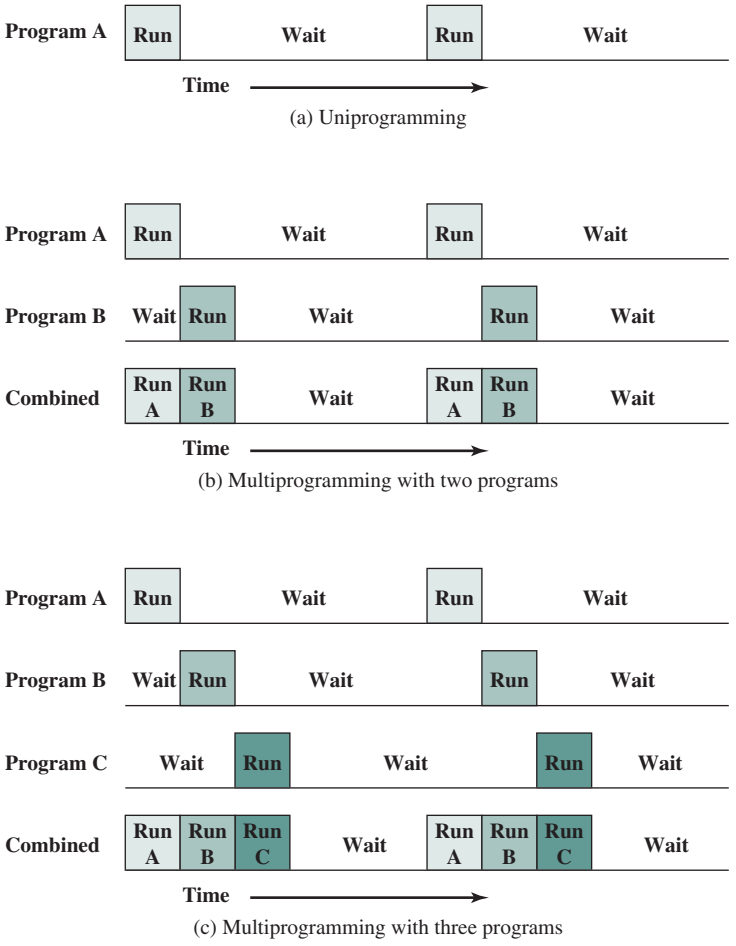


Figure 8.5 Multiprogramming Example

time executing, until it reaches an I/O instruction. It must then wait until that I/O instruction concludes before proceeding.

This inefficiency is not necessary. We know that there must be enough memory to hold the OS (resident monitor) and one user program. Suppose that there is room for the OS and two user programs. Now, when one job needs to wait for I/O, the processor can switch to the other job, which likely is not waiting for I/O (Figure 8.5b). Furthermore, we might expand memory to hold three, four, or more programs and switch among all of them (Figure 8.5c). This technique is known as **multiprogramming**, or **multitasking**.[1] It is the central theme of modern operating systems.

[1]The term *multitasking* is sometimes reserved to mean multiple tasks within the same program that may be handled concurrently by the OS, in contrast to *multiprogramming,* which would refer to multiple processes from multiple programs. However, it is more common to equate the terms *multitasking* and *multiprogramming,* as is done in most standards dictionaries (e.g., IEEE Std 100-1992, *The New IEEE Standard Dictionary of Electrical and Electronics Terms*).

EXAMPLE 8.1 This example illustrates the benefit of multiprogramming. Consider a computer with 250 Mbytes of available memory (not used by the OS), a disk, a terminal, and a printer. Three programs, JOB1, JOB2, and JOB3, are submitted for execution at the same time, with the attributes listed in Table 8.1. We assume minimal processor requirements for JOB1 and JOB2 and continuous disk and printer use by JOB3. For a simple batch environment, these jobs will be executed in sequence. Thus, JOB1 completes in 5 minutes. JOB2 must wait until the 5 minutes is over and then completes 15 minutes after that. JOB3 begins after 20 minutes and completes at 30 minutes from the time it was initially submitted. The average resource utilization, throughput, and response times are shown in the uniprogramming column of Table 8.2. Device-by-device utilization is illustrated in Figure 8.6a. It is evident that there is gross underutilization for all resources when averaged over the required 30-minute time period.

Now suppose that the jobs are run concurrently under a multiprogramming OS. Because there is little resource contention between the jobs, all three can run in nearly minimum time while coexisting with the others in the computer (assuming that JOB2 and JOB3 are allotted enough processor time to keep their input and output operations active). JOB1 will still require 5 minutes to complete but at the end of that time, JOB2 will be one-third finished, and JOB3 will be half finished. All three jobs will have finished within 15 minutes. The improvement is evident when examining the multiprogramming column of Table 8.2, obtained from the histogram shown in Figure 8.6b.

As with a simple batch system, a multiprogramming batch system must rely on certain computer hardware features. The most notable additional feature that is useful for multiprogramming is the hardware that supports I/O interrupts

Table 8.1 Sample Program Execution Attributes

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration (min)	5	15	10
Memory required (M)	50	100	80
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

Table 8.2 Effects of Multiprogramming on Resource Utilization

	Uniprogramming	Multiprogramming
Processor use (%)	20	40
Memory use (%)	33	67
Disk use (%)	33	67
Printer use (%)	33	67
Elapsed time (min)	30	15
Throughput rate (jobs/hr)	6	12
Mean response time (min)	18	10

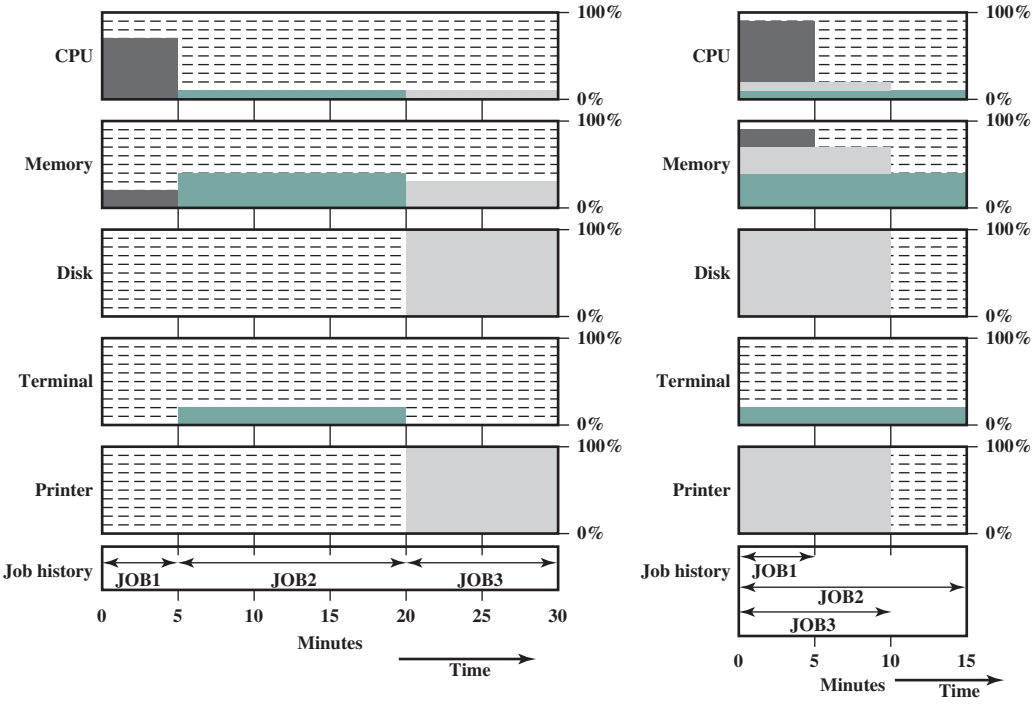


Figure 8.6 Utilization Histograms

and DMA. With interrupt-driven I/O or DMA, the processor can issue an I/O command for one job and proceed with the execution of another job while the I/O is carried out by the device controller. When the I/O operation is complete, the processor is interrupted and control is passed to an interrupt-handling program in the OS. The OS will then pass control to another job.

Multiprogramming operating systems are fairly sophisticated compared to single-program, or **uniprogramming**, systems. To have several jobs ready to run, the jobs must be kept in main memory, requiring some form of **memory management**. In addition, if several jobs are ready to run, the processor must decide which one to run, which requires some algorithm for scheduling. These concepts are discussed later in this chapter.

TIME-SHARING SYSTEMS With the use of multiprogramming, batch processing can be quite efficient. However, for many jobs, it is desirable to provide a mode in which the user interacts directly with the computer. Indeed, for some jobs, such as transaction processing, an interactive mode is essential.

Today, the requirement for an interactive computing facility can be, and often is, met by the use of a dedicated microcomputer. That option was not available in the 1960s, when most computers were big and costly. Instead, time sharing was developed.

Just as multiprogramming allows the processor to handle multiple batch jobs at a time, multiprogramming can be used to handle multiple interactive jobs. In this latter case, the technique is referred to as time sharing, because the processor's time is shared among multiple users. In a **time-sharing system**, multiple users

Table 8.3 Batch Multiprogramming versus Time Sharing

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

simultaneously access the system through terminals, with the OS interleaving the execution of each user program in a short burst or quantum of computation. Thus, if there are n users actively requesting service at one time, each user will only see on the average $1/n$ of the effective computer speed, not counting OS overhead. However, given the relatively slow human reaction time, the response time on a properly designed system should be comparable to that on a dedicated computer.

Both batch multiprogramming and time sharing use multiprogramming. The key differences are listed in Table 8.3.

8.2 SCHEDULING

The key to multiprogramming is scheduling. In fact, four types of scheduling are typically involved (Table 8.4). We will explore these presently. But first, we introduce the concept of **process**. This term was first used by the designers of the Multics OS in the 1960s. It is a somewhat more general term than *job*. Many definitions have been given for the term *process*, including

- A program in execution
- The "animated spirit" of a program
- That entity to which a processor is assigned

This concept should become clearer as we proceed.

Long-Term Scheduling

The long-term scheduler determines which programs are admitted to the system for processing. Thus, it controls the degree of multiprogramming (number of processes in memory). Once admitted, a job or user program becomes a process and is added to the queue for the short-term scheduler. In some systems, a newly created process begins in a swapped-out condition, in which case it is added to a queue for the medium-term scheduler.

Table 8.4 Types of Scheduling

Long-term scheduling	The decision to add to the pool of processes to be executed.
Medium-term scheduling	The decision to add to the number of processes that are partially or fully in main memory.
Short-term scheduling	The decision as to which available process will be executed by the processor.
I/O scheduling	The decision as to which process's pending I/O request shall be handled by an available I/O device.

In a batch system, or for the batch portion of a general-purpose OS, newly submitted jobs are routed to disk and held in a batch queue. The long-term scheduler creates processes from the queue when it can. There are two decisions involved here. First, the scheduler must decide that the OS can take on one or more additional processes. Second, the scheduler must decide which job or jobs to accept and turn into processes. The criteria used may include priority, expected execution time, and I/O requirements.

For interactive programs in a time-sharing system, a process request is generated when a user attempts to connect to the system. Time-sharing users are not simply queued up and kept waiting until the system can accept them. Rather, the OS will accept all authorized comers until the system is saturated, using some predefined measure of saturation. At that point, a connection request is met with a message indicating that the system is full and the user should try again later.

Medium-Term Scheduling

Medium-term scheduling is part of the swapping function, described in Section 8.3. Typically, the swapping-in decision is based on the need to manage the degree of multiprogramming. On a system that does not use virtual memory, memory management is also an issue. Thus, the swapping-in decision will consider the memory requirements of the swapped-out processes.

Short-Term Scheduling

The long-term scheduler executes relatively infrequently and makes the coarse-grained decision of whether or not to take on a new process, and which one to take. The short-term scheduler, also known as the **dispatcher**, executes frequently and makes the fine-grained decision of which job to execute next.

PROCESS STATES To understand the operation of the short-term scheduler, we need to consider the concept of a **process state**. During the lifetime of a process, its status will change a number of times. Its status at any point in time is referred to as a *state*. The term *state* is used because it connotes that certain information exists that defines the status at that point. At minimum, there are five defined states for a process (Figure 8.7):

- **New:** A program is admitted by the high-level scheduler but is not yet ready to execute. The OS will initialize the process, moving it to the ready state.

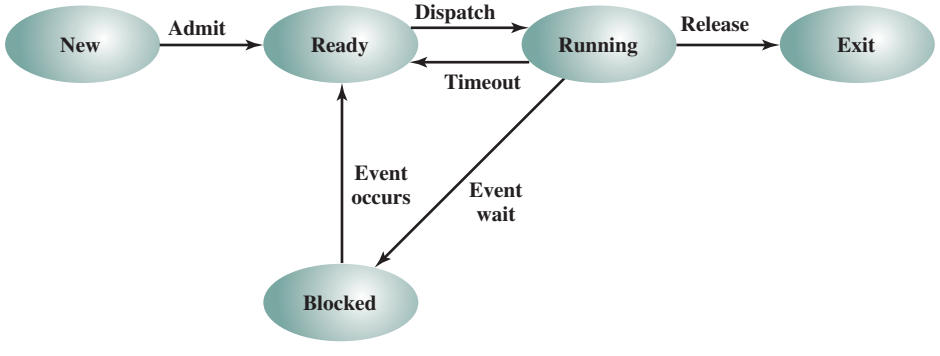


Figure 8.7 Five-State Process Model

- **Ready:** The process is ready to execute and is awaiting access to the processor.
- **Running:** The process is being executed by the processor.
- **Waiting:** The process is suspended from execution waiting for some system resource, such as I/O.
- **Halted:** The process has terminated and will be destroyed by the OS.

For each process in the system, the OS must maintain information indicating the state of the process and other information necessary for process execution. For this purpose, each process is represented in the OS by a **process control block** (Figure 8.8), which typically contains:

- **Identifier:** Each current process has a unique identifier.
- **State:** The current state of the process (new, ready, and so on).
- **Priority:** Relative priority level.
- **Program counter:** The address of the next instruction in the program to be executed.
- **Memory pointers:** The starting and ending locations of the process in memory.
- **Context data:** These are data that are present in registers in the processor while the process is executing, and they will be discussed in Part Three. For now, it is enough to say that these data represent the "context" of the process. The context data plus the program counter are saved when the process leaves the running state. They are retrieved by the processor when it resumes execution of the process.

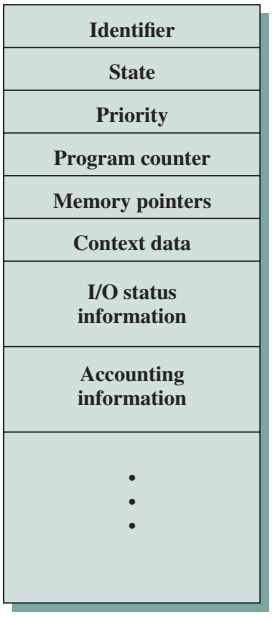


Figure 8.8 Process Control Block

- **I/O status information:** Includes outstanding I/O requests, I/O devices (e.g., tape drives) assigned to this process, a list of files assigned to the process, and so on.
- **Accounting information:** May include the amount of processor time and clock time used, time limits, account numbers, and so on.

When the scheduler accepts a new job or user request for execution, it creates a blank process control block and places the associated process in the new state. After the system has properly filled in the process control block, the process is transferred to the ready state.

SCHEDULING TECHNIQUES To understand how the OS manages the scheduling of the various jobs in memory, let us begin by considering the simple example in Figure 8.9. The figure shows how main memory is partitioned at a given point in time. The kernel of the OS is, of course, always resident. In addition, there are a number of active processes, including **A** and **B**, each of which is allocated a portion of memory.

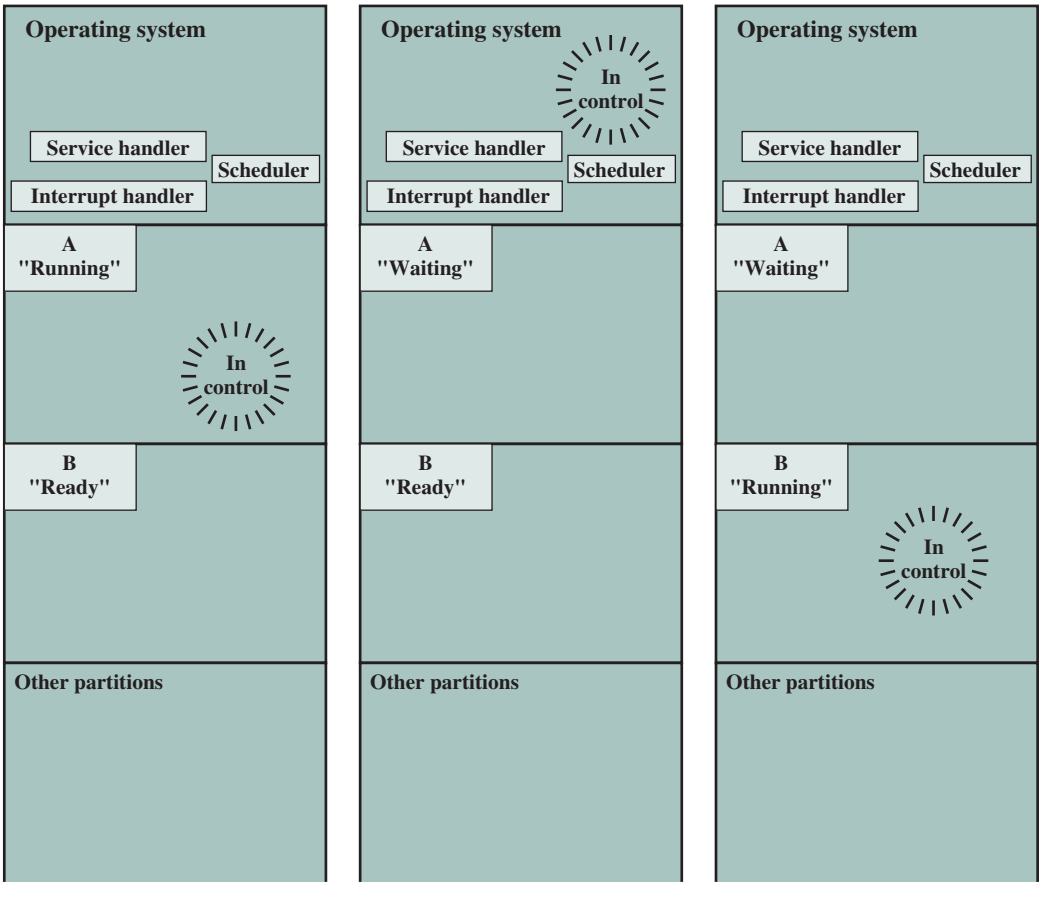


Figure 8.9 Scheduling Example

We begin at a point in time when process **A** is running. The processor is executing instructions from the program contained in **A**'s memory partition. At some later point in time, the processor ceases to execute instructions in **A** and begins executing instructions in the OS area. This will happen for one of three reasons:

1. Process **A** issues a service call (e.g., an I/O request) to the OS. Execution of **A** is suspended until this call is satisfied by the OS.
2. Process **A** causes an *interrupt.* An interrupt is a hardware-generated signal to the processor. When this signal is detected, the processor ceases to execute **A** and transfers to the interrupt handler in the OS. A variety of events related to **A** will cause an interrupt. One example is an error, such as attempting to execute a privileged instruction. Another example is a timeout; to prevent any one process from monopolizing the processor, each process is only granted the processor for a short period at a time.
3. Some event unrelated to process **A** that requires attention causes an interrupt. An example is the completion of an I/O operation.

In any case, the result is the following. The processor saves the current context data and the program counter for **A** in **A**'s process control block and then begins executing in the OS. The OS may perform some work, such as initiating an I/O operation. Then the short-term-scheduler portion of the OS decides which process should be executed next. In this example, **B** is chosen. The OS instructs the processor to restore **B**'s context data and proceed with the execution of **B** where it left off.

This simple example highlights the basic functioning of the short-term scheduler. Figure 8.10 shows the major elements of the OS involved in the multiprogramming and scheduling of processes. The OS receives control of the processor at the

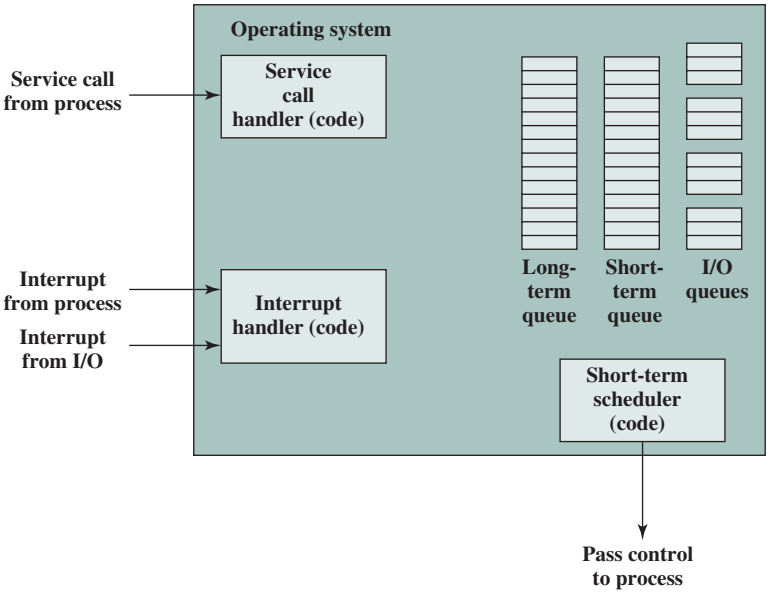


Figure 8.10 Key Elements of an Operating System for Multiprogramming

interrupt handler if an interrupt occurs and at the service-call handler if a service call occurs. Once the interrupt or service call is handled, the short-term scheduler is invoked to select a process for execution.

To do its job, the OS maintains a number of queues. Each queue is simply a waiting list of processes waiting for some resource. The **long-term queue** is a list of jobs waiting to use the system. As conditions permit, the high-level scheduler will allocate memory and create a process for one of the waiting items. The **short-term queue** consists of all processes in the ready state. Any one of these processes could use the processor next. It is up to the short-term scheduler to pick one. Generally, this is done with a round-robin algorithm, giving each process some time in turn. Priority levels may also be used. Finally, there is an **I/O queue** for each I/O device. More than one process may request the use of the same I/O device. All processes waiting to use each device are lined up in that device's queue.

Figure 8.11 suggests how processes progress through the computer under the control of the OS. Each process request (batch job, user-defined interactive job) is placed in the long-term queue. As resources become available, a process request becomes a process and is then placed in the ready state and put in the short-term queue. The processor alternates between executing OS instructions and executing user processes. While the OS is in control, it decides which process in the short-term queue should be executed next. When the OS has finished its immediate tasks, it turns the processor over to the chosen process.

As was mentioned earlier, a process being executed may be suspended for a variety of reasons. If it is suspended because the process requests I/O, then it

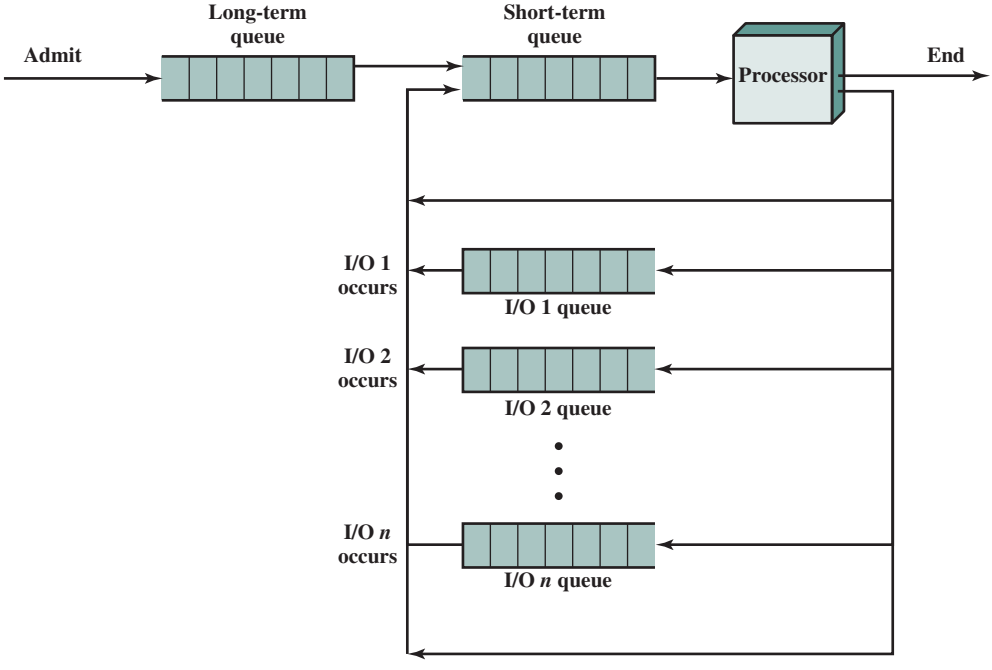


Figure 8.11 Queuing Diagram Representation of Processor Scheduling

is placed in the appropriate I/O queue. If it is suspended because of a timeout or because the OS must attend to pressing business, then it is placed in the ready state and put into the short-term queue.

Finally, we mention that the OS also manages the I/O queues. When an I/O operation is completed, the OS removes the satisfied process from that I/O queue and places it in the short-term queue. It then selects another waiting process (if any) and signals for the I/O device to satisfy that process's request.

8.3 MEMORY MANAGEMENT

In a uniprogramming system, main memory is divided into two parts: one part for the OS (resident monitor) and one part for the program currently being executed. In a multiprogramming system, the "user" part of memory is subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the OS and is known as **memory management**.

Effective memory management is vital in a multiprogramming system. If only a few processes are in memory, then for much of the time all of the processes will be waiting for I/O and the processor will be idle. Thus, memory needs to be allocated efficiently to pack as many processes into memory as possible.

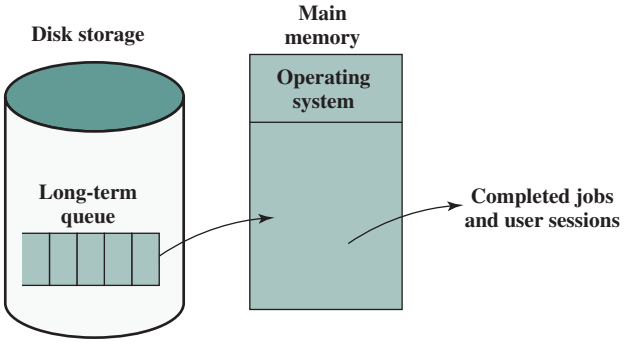
Swapping

Referring back to Figure 8.11, we have discussed three types of queues: the long-term queue of requests for new processes, the short-term queue of processes ready to use the processor, and the various I/O queues of processes that are not ready to use the processor. Recall that the reason for this elaborate machinery is that I/O activities are much slower than computation and therefore the processor in a uniprogramming system is idle most of the time.

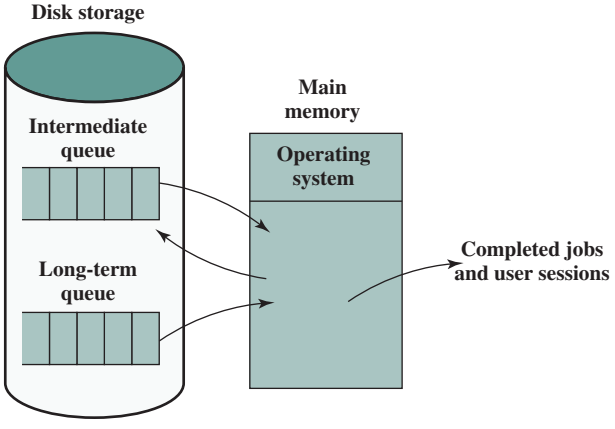
But the arrangement in Figure 8.11 does not entirely solve the problem. It is true that, in this case, memory holds multiple processes and that the processor can move to another process when one process is waiting. But the processor is so much faster than I/O that it will be common for *all* the processes in memory to be waiting on I/O. Thus, even with multiprogramming, a processor could be idle most of the time.

What to do? Main memory could be expanded, and so be able to accommodate more processes. But there are two flaws in this approach. First, main memory is expensive, even today. Second, the appetite of programs for memory has grown as fast as the cost of memory has dropped. So larger memory results in larger processes, not more processes.

Another solution is **swapping**, depicted in Figure 8.12. We have a long-term queue of process requests, typically stored on disk. These are brought in, one at a time, as space becomes available. As processes are completed, they are moved out of main memory. Now the situation will arise that none of the processes in memory are in the ready state (e.g., all are waiting on an I/O operation). Rather than remain idle, the processor *swaps* one of these processes back out to disk into an *intermediate queue*. This is a queue of existing processes that have been temporarily



(a) Simple job scheduling



(b) Swapping

Figure 8.12 The Use of Swapping

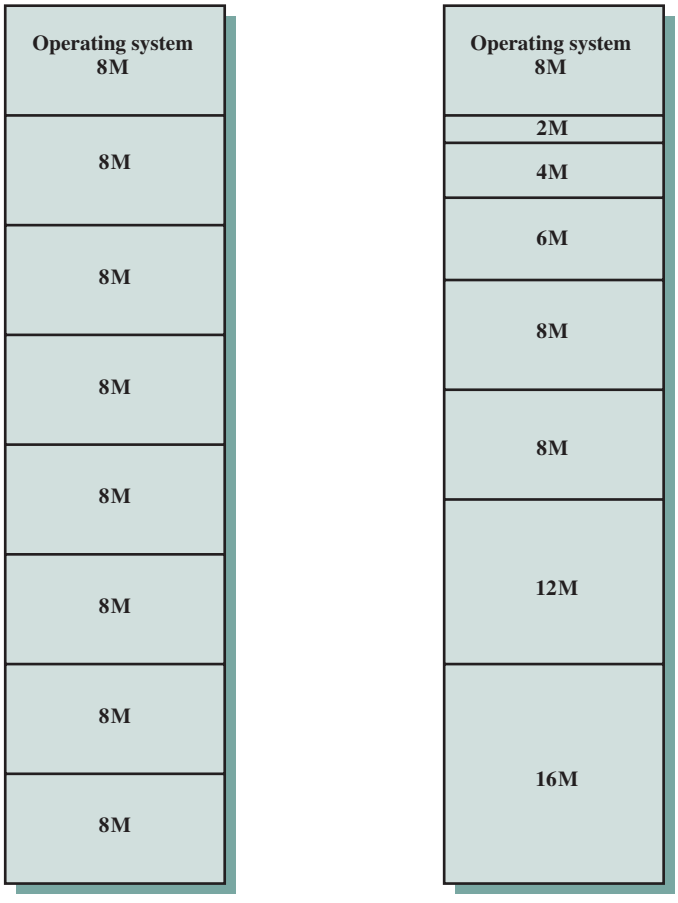
kicked out of memory. The OS then brings in another process from the intermediate queue, or it honors a new process request from the long-term queue. Execution then continues with the newly arrived process.

Swapping, however, is an I/O operation, and therefore there is the potential for making the problem worse, not better. But because disk I/O is generally the fastest I/O on a system (e.g., compared with tape or printer I/O), swapping will usually enhance performance. A more sophisticated scheme, involving virtual memory, improves performance over simple swapping. This will be discussed shortly. But first, we must prepare the ground by explaining partitioning and paging.

Partitioning

The simplest scheme for partitioning available memory is to use *fixed-size partitions,* as shown in Figure 8.13. Note that, although the partitions are of fixed size, they need not be of equal size. When a process is brought into memory, it is placed in the smallest available partition that will hold it.

Even with the use of unequal fixed-size partitions, there will be wasted memory. In most cases, a process will not require exactly as much memory as provided



(a) Equal-size partitions (b) Unequal-size partitions

Figure 8.13 Example of Fixed Partitioning of a 64-Mbyte Memory

by the partition. For example, a process that requires 3M bytes of memory would be placed in the 4M partition of Figure 8.13b, wasting 1M that could be used by another process.

A more efficient approach is to use *variable-size partitions.* When a process is brought into memory, it is allocated exactly as much memory as it requires and no more.

EXAMPLE 8.2 An example, using 64 Mbytes of main memory, is shown in Figure 8.14. Initially, main memory is empty, except for the OS (a). The first three processes are loaded in, starting where the OS ends and occupying just enough space for each process (b, c, d). This leaves a "hole" at the end of memory that is too small for a fourth process. At some point, none of the processes in memory is ready. The OS swaps out process 2 (e), which leaves sufficient room to load a new process, process 4 (f). Because process 4 is smaller than process 2, another small hole is created. Later, a point is reached at which none of the processes in main memory is ready, but process 2, in the ready-suspend state, is available. Because there is insufficient room in memory for process 2, the OS swaps process 1 out (g) and swaps process 2 back in (h).

As this example shows, this method starts out well, but eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on, memory becomes more and more fragmented, and memory utilization declines. One technique for overcoming this problem is **compaction**: From time to time, the OS shifts the processes in memory to place all the free memory together in one block. This is a time-consuming procedure, wasteful of processor time.

Before we consider ways of dealing with the shortcomings of partitioning, we must clear up one loose end. Consider Figure 8.14; it should be obvious that a process is not likely to be loaded into the same place in main memory each time it is swapped in. Furthermore, if compaction is used, a process may be shifted while in main memory. A process in memory consists of instructions plus data. The instructions will contain addresses for memory locations of two types:

- Addresses of data items
- Addresses of instructions, used for branching instructions

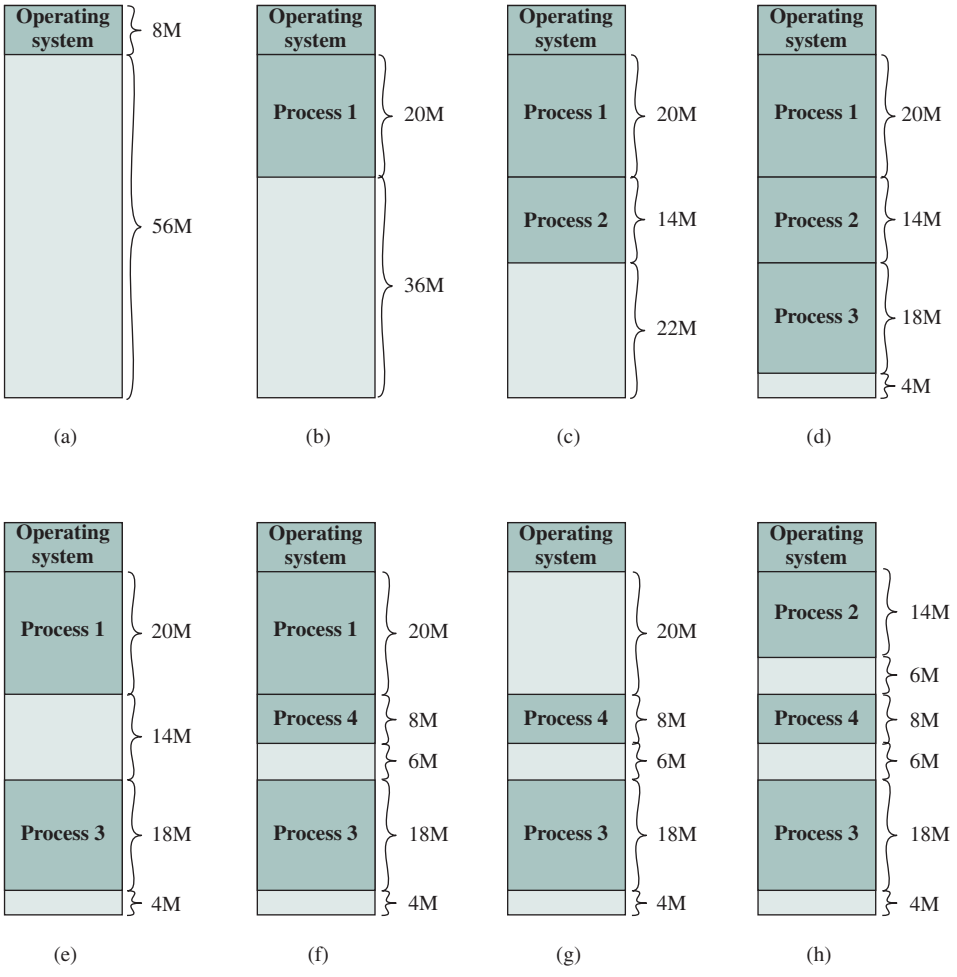


Figure 8.14 The Effect of Dynamic Partitioning

But these addresses are not fixed. They will change each time a process is swapped in. To solve this problem, a distinction is made between logical addresses and physical addresses. A **logical address** is expressed as a location relative to the beginning of the program. Instructions in the program contain only logical addresses. A **physical address** is an actual location in main memory. When the processor executes a process, it automatically converts from logical to physical address by adding the current starting location of the process, called its **base address**, to each logical address. This is another example of a processor hardware feature designed to meet an OS requirement. The exact nature of this hardware feature depends on the memory management strategy in use. We will see several examples later in this chapter.

Paging

Both unequal fixed-size and variable-size partitions are inefficient in the use of memory. Suppose, however, that memory is partitioned into equal fixed-size chunks that are relatively small, and that each process is also divided into small fixed-size chunks of some size. Then the chunks of a program, known as **pages**, could be assigned to available chunks of memory, known as **frames**, or page frames. At most, then, the wasted space in memory for that process is a fraction of the last page.

Figure 8.15 shows an example of the use of pages and frames. At a given point in time, some of the frames in memory are in use and some are free. The list of free frames is maintained by the OS. Process **A**, stored on disk, consists of four pages.

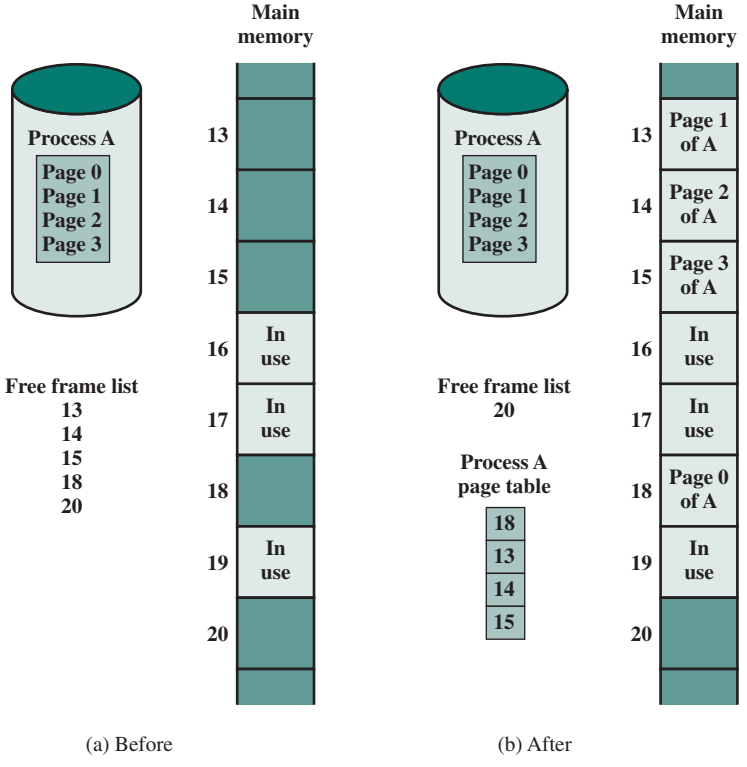


Figure 8.15 Allocation of Free Frames

When it comes time to load this process, the OS finds four free frames and loads the four pages of the process **A** into the four frames.

Now suppose, as in this example, that there are not sufficient unused contiguous frames to hold the process. Does this prevent the OS from loading **A**? The answer is no, because we can once again use the concept of logical address. A simple base address will no longer suffice. Rather, the OS maintains a **page table** for each process. The page table shows the frame location for each page of the process. Within the program, each logical address consists of a page number and a relative address within the page. Recall that in the case of simple partitioning, a logical address is the location of a word relative to the beginning of the program; the processor translates that into a physical address. With paging, the logical-to-physical address translation is still done by processor hardware. The processor must know how to access the page table of the current process. Presented with a logical address (page number, relative address), the processor uses the page table to produce a physical address (frame number, relative address). An example is shown in Figure 8.16.

This approach solves the problems raised earlier. Main memory is divided into many small equal-size frames. Each process is divided into frame-size pages: smaller processes require fewer pages, larger processes require more. When a process is brought in, its pages are loaded into available frames, and a page table is set up.

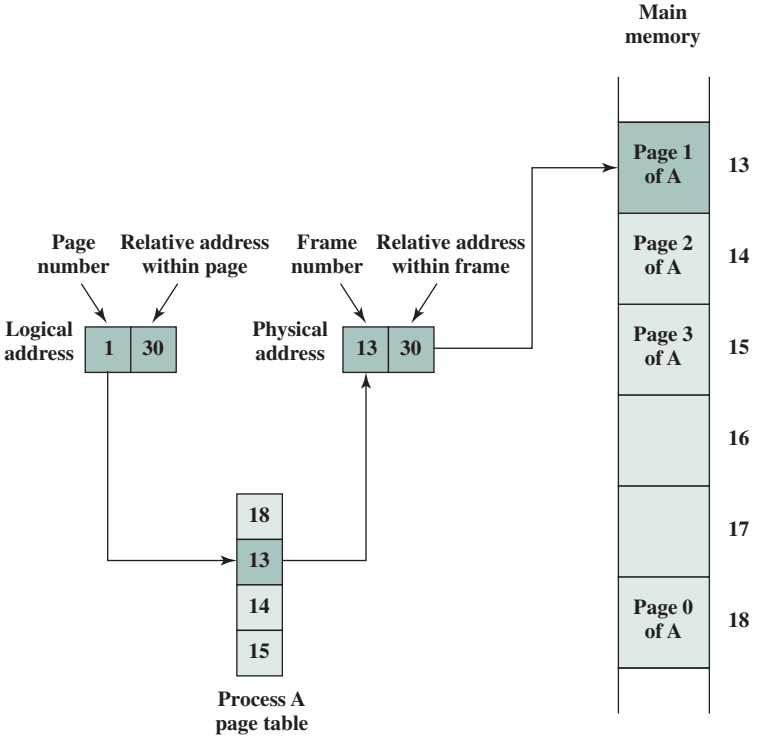


Figure 8.16 Logical and Physical Addresses

Virtual Memory

DEMAND PAGING With the use of paging, truly effective multiprogramming systems came into being. Furthermore, the simple tactic of breaking a process up into pages led to the development of another important concept: virtual memory.

To understand virtual memory, we must add a refinement to the paging scheme just discussed. That refinement is **demand paging**, which simply means that each page of a process is brought in only when it is needed, that is, on demand.

Consider a large process, consisting of a long program plus a number of arrays of data. Over any short period of time, execution may be confined to a small section of the program (e.g., a subroutine), and perhaps only one or two arrays of data are being used. This is the principle of locality, which we introduced in Appendix 4A. It would clearly be wasteful to load in dozens of pages for that process when only a few pages will be used before the program is suspended. We can make better use of memory by loading in just a few pages. Then, if the program branches to an instruction on a page not in main memory, or if the program references data on a page not in memory, a **page fault** is triggered. This tells the OS to bring in the desired page.

Thus, at any one time, only a few pages of any given process are in memory, and therefore more processes can be maintained in memory. Furthermore, time is saved because unused pages are not swapped in and out of memory. However, the OS must be clever about how it manages this scheme. When it brings one page in, it must throw another page out; this is known as **page replacement**. If it throws out a page just before it is about to be used, then it will just have to go get that page again almost immediately. Too much of this leads to a condition known as **thrashing**: the processor spends most of its time swapping pages rather than executing instructions. The avoidance of thrashing was a major research area in the 1970s and led to a variety of complex but effective algorithms. In essence, the OS tries to guess, based on recent history, which pages are least likely to be used in the near future.

Page Replacement Algorithm Simulators

A discussion of page replacement algorithms is beyond the scope of this chapter. A potentially effective technique is least recently used (LRU), the same algorithm discussed in Chapter 4 for cache replacement. In practice, LRU is difficult to implement for a virtual memory paging scheme. Several alternative approaches that seek to approximate the performance of LRU are in use; see Appendix K for details.

With demand paging, it is not necessary to load an entire process into main memory. This fact has a remarkable consequence: *It is possible for a process to be larger than all of main memory.* One of the most fundamental restrictions in programming has been lifted. Without demand paging, a programmer must be acutely aware of how much memory is available. If the program being written is too large, the programmer must devise ways to structure the program into pieces that can be

loaded one at a time. With demand paging, that job is left to the OS and the hardware. As far as the programmer is concerned, he or she is dealing with a huge memory, the size associated with disk storage.

Because a process executes only in main memory, that memory is referred to as **real memory**. But a programmer or user perceives a much larger memory—that which is allocated on the disk. This latter is therefore referred to as **virtual memory**. Virtual memory allows for very effective multiprogramming and relieves the user of the unnecessarily tight constraints of main memory.

PAGE TABLE STRUCTURE The basic mechanism for reading a word from memory involves the translation of a virtual, or logical, address, consisting of page number and offset, into a physical address, consisting of frame number and offset, using a page table. Because the page table is of variable length, depending on the size of the process, we cannot expect to hold it in registers. Instead, it must be in main memory to be accessed. Figure 8.16 suggests a hardware implementation of this scheme. When a particular process is running, a register holds the starting address of the page table for that process. The page number of a virtual address is used to index that table and look up the corresponding frame number. This is combined with the offset portion of the virtual address to produce the desired real address.

In most systems, there is one page table per process. But each process can occupy huge amounts of virtual memory. For example, in the VAX architecture, each process can have up to $2^{31} = 2$ Gbytes of virtual memory. Using $2^9 = 512 -$ byte pages, that means that as many as 2^{22} page table entries are required *per process.* Clearly, the amount of memory devoted to page tables alone could be unacceptably high. To overcome this problem, most virtual memory schemes store page tables in virtual memory rather than real memory. This means that page tables are subject to paging just as other pages are. When a process is running, at least a part of its page table must be in main memory, including the page table entry of the currently executing page. Some processors make use of a two-level scheme to organize large page tables. In this scheme, there is a page directory, in which each entry points to a page table. Thus, if the length of the page directory is X, and if the maximum length of a page table is Y, then a process can consist of up to $X \times Y$ pages. Typically, the maximum length of a page table is restricted to be equal to one page. We will see an example of this two-level approach when we consider the Intel x86 later in this chapter.

An alternative approach to the use of one- or two-level page tables is the use of an inverted page table structure (Figure 8.17). Variations on this approach are used on the PowerPC, UltraSPARC, and the IA-64 architecture. An implementation of the Mach OS on the RT-PC also uses this technique.

In this approach, the page number portion of a virtual address is mapped into a hash value using a simple hashing function.[2] The hash value is a pointer to the inverted page table, which contains the page table entries. There is one entry in the

[2]A hash function maps numbers in the range 0 through M into numbers in the range 0 through N, where $M > N$. The output of the hash function is used as an index into the hash table. Since more than one input maps into the same output, it is possible for an input item to map to a hash table entry that is already occupied. In that case, the new item must *overflow* into another hash table location. Typically, the new item is placed in the first succeeding empty space, and a pointer from the original location is provided to chain the entries together. See Appendix L for more information on hash functions.

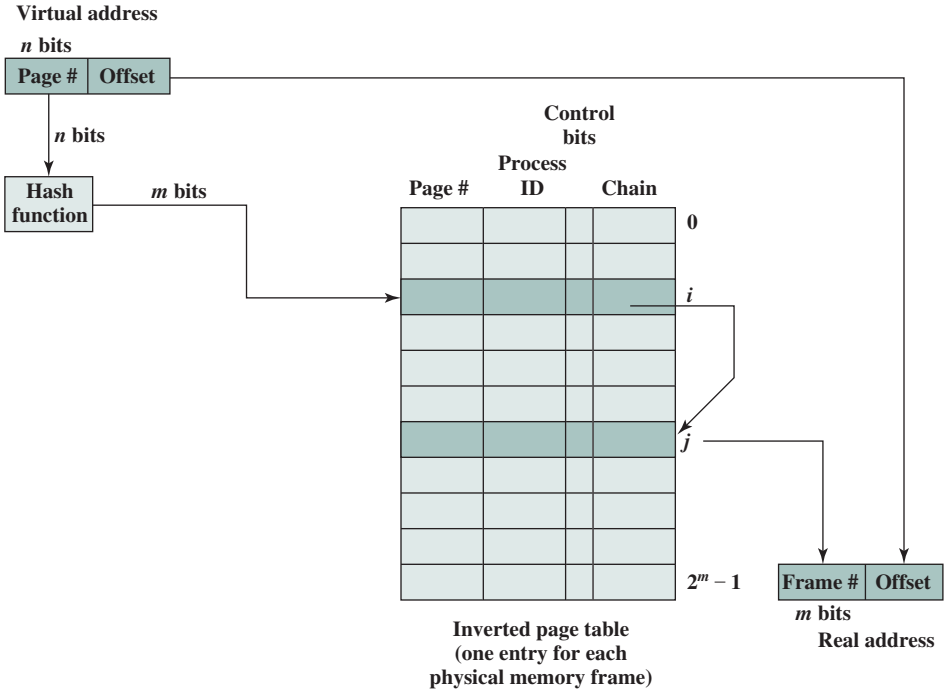


Figure 8.17 Inverted Page Table Structure

inverted page table for each real memory page frame rather than one per virtual page. Thus a fixed proportion of real memory is required for the tables regardless of the number of processes or virtual pages supported. Because more than one virtual address may map into the same hash table entry, a chaining technique is used for managing the overflow. The hashing technique results in chains that are typically short—between one and two entries. The page table's structure is called *inverted* because it indexes page table entries by frame number rather than by virtual page number.

Translation Lookaside Buffer

In principle, then, every virtual memory reference can cause two physical memory accesses: one to fetch the appropriate page table entry, and one to fetch the desired data. Thus, a straightforward virtual memory scheme would have the effect of doubling the memory access time. To overcome this problem, most virtual memory schemes make use of a special cache for page table entries, usually called a **translation lookaside buffer (TLB)**. This cache functions in the same way as a memory cache and contains those page table entries that have been most recently used. Figure 8.18 is a flowchart that shows the use of the TLB. By the principle of locality, most virtual memory references will be to locations in recently used pages. Therefore, most references will involve page table entries in the cache. Studies of the VAX TLB have shown that this scheme can significantly improve performance [CLAR85, SATY81].

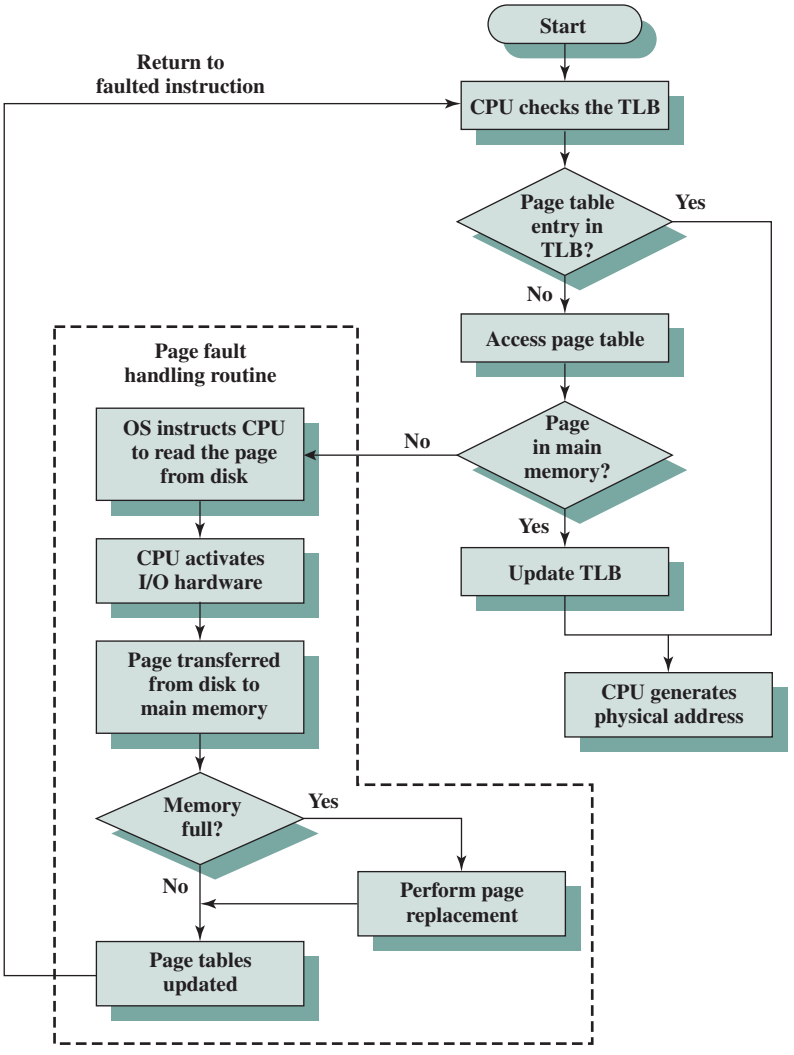


Figure 8.18 Operation of Paging and Translation Lookaside Buffer (TLB)

Note that the virtual memory mechanism must interact with the cache system (not the TLB cache, but the main memory cache). This is illustrated in Figure 8.19. A virtual address will generally be in the form of a page number, offset. First, the memory system consults the TLB to see if the matching page table entry is present. If it is, the real (physical) address is generated by combining the frame number with the offset. If not, the entry is accessed from a page table. Once the real address is generated, which is in the form of a tag and a remainder, the cache is consulted to see if the block containing that word is present (see Figure 4.5). If so, it is returned to the processor. If not, the word is retrieved from main memory.

The reader should be able to appreciate the complexity of the processor hardware involved in a single memory reference. The virtual address is translated into a real address. This involves reference to a page table, which may be in the TLB, in

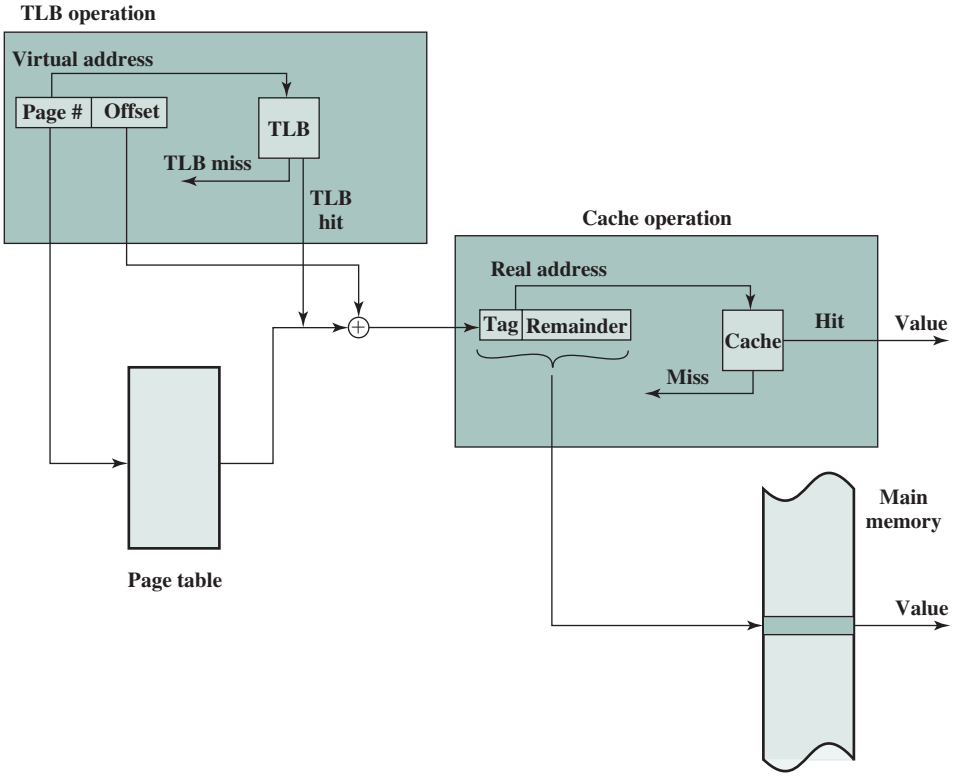


Figure 8.19 Translation Lookaside Buffer and Cache Operation

main memory, or on disk. The referenced word may be in cache, in main memory, or on disk. In the latter case, the page containing the word must be loaded into main memory and its block loaded into the cache. In addition, the page table entry for that page must be updated.

Segmentation

There is another way in which addressable memory can be subdivided, known as *segmentation*. Whereas paging is invisible to the programmer and serves the purpose of providing the programmer with a larger address space, segmentation is usually visible to the programmer and is provided as a convenience for organizing programs and data and as a means for associating privilege and protection attributes with instructions and data.

Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments. Segments are of variable, indeed dynamic, size. Typically, the programmer or the OS will assign programs and data to different segments. There may be a number of program segments for various types of programs as well as a number of data segments. Each segment may be assigned access and usage rights. Memory references consist of a (segment number, offset) form of address.

This organization has a number of advantages to the programmer over a non-segmented address space:

1. It simplifies the handling of growing data structures. If the programmer does not know ahead of time how large a particular data structure will become, it is not necessary to guess. The data structure can be assigned its own segment, and the OS will expand or shrink the segment as needed.
2. It allows programs to be altered and recompiled independently without requiring that an entire set of programs be relinked and reloaded. Again, this is accomplished using multiple segments.
3. It lends itself to sharing among processes. A programmer can place a utility program or a useful table of data in a segment that can be addressed by other processes.
4. It lends itself to protection. Because a segment can be constructed to contain a well-defined set of programs or data, the programmer or a system administrator can assign access privileges in a convenient fashion.

These advantages are not available with paging, which is invisible to the programmer. On the other hand, we have seen that paging provides for an efficient form of memory management. To combine the advantages of both, some systems are equipped with the hardware and OS software to provide both.

8.4 INTEL x86 MEMORY MANAGEMENT

Since the introduction of the 32-bit architecture, microprocessors have evolved sophisticated memory management schemes that build on the lessons learned with medium- and large-scale systems. In many cases, the microprocessor versions are superior to their larger-system antecedents. Because the schemes were developed by the microprocessor hardware vendor and may be employed with a variety of operating systems, they tend to be quite general purpose. A representative example is the scheme used on the Intel x86 architecture.

Address Spaces

The x86 includes hardware for both segmentation and paging. Both mechanisms can be disabled, allowing the user to choose from four distinct views of memory:

- **Unsegmented unpaged memory:** In this case, the virtual address is the same as the physical address. This is useful, for example, in low-complexity, high-performance controller applications.
- **Unsegmented paged memory:** Here memory is viewed as a paged linear address space. Protection and management of memory is done via paging. This is favored by some operating systems (e.g., Berkeley UNIX).
- **Segmented unpaged memory:** Here memory is viewed as a collection of logical address spaces. The advantage of this view over a paged approach is that it affords protection down to the level of a single byte, if necessary. Furthermore, unlike paging, it guarantees that the translation table needed (the segment table) is on-chip when the segment is in memory. Hence, segmented unpaged memory results in predictable access times.

- **Segmented paged memory:** Segmentation is used to define logical memory partitions subject to access control, and paging is used to manage the allocation of memory within the partitions. Operating systems such as UNIX System V favor this view.

Segmentation

When segmentation is used, each virtual address (called a logical address in the x86 documentation) consists of a 16-bit segment reference and a 32-bit offset. Two bits of the segment reference deal with the protection mechanism, leaving 14 bits for specifying a particular segment. Thus, with unsegmented memory, the user's virtual memory is $2^{32} = 4$ Gbytes. With segmented memory, the total virtual memory space as seen by a user is $2^{46} = 64$ terabytes (Tbytes). The physical address space employs a 32-bit address for a maximum of 4 Gbytes.

The amount of virtual memory can actually be larger than the 64 Tbytes. This is because the processor's interpretation of a virtual address depends on which process is currently active. Virtual address space is divided into two parts. One-half of the virtual address space (8K segments $\times$ 4 Gbytes) is global, shared by all processes; the remainder is local and is distinct for each process.

Associated with each segment are two forms of protection: privilege level and access attribute. There are four privilege levels, from most protected (level 0) to least protected (level 3). The privilege level associated with a data segment is its "classification"; the privilege level associated with a program segment is its "clearance." An executing program may only access data segments for which its clearance level is lower than (more privileged) or equal to (same privilege) the privilege level of the data segment.

The hardware does not dictate how these privilege levels are to be used; this depends on the OS design and implementation. It was intended that privilege level 1 would be used for most of the OS, and level 0 would be used for that small portion of the OS devoted to memory management, protection, and access control. This leaves two levels for applications. In many systems, applications will reside at level 3, with level 2 being unused. Specialized application subsystems that must be protected because they implement their own security mechanisms are good candidates for level 2. Some examples are database management systems, office automation systems, and software engineering environments.

In addition to regulating access to data segments, the privilege mechanism limits the use of certain instructions. Some instructions, such as those dealing with memory-management registers, can only be executed in level 0. I/O instructions can only be executed up to a certain level that is designated by the OS; typically, this will be level 1.

The access attribute of a data segment specifies whether read/write or read-only accesses are permitted. For program segments, the access attribute specifies read/execute or read-only access.

The address translation mechanism for segmentation involves mapping a virtual address into what is referred to as a linear address (Figure 8.20b). A virtual address consists of the 32-bit offset and a 16-bit segment selector (Figure 8.20a). An instruction fetching or storing an operand specifies the offset and a register containing the segment selector. The segment selector consists of the following fields:

- **Table Indicator (TI):** Indicates whether the global segment table or a local segment table should be used for translation.

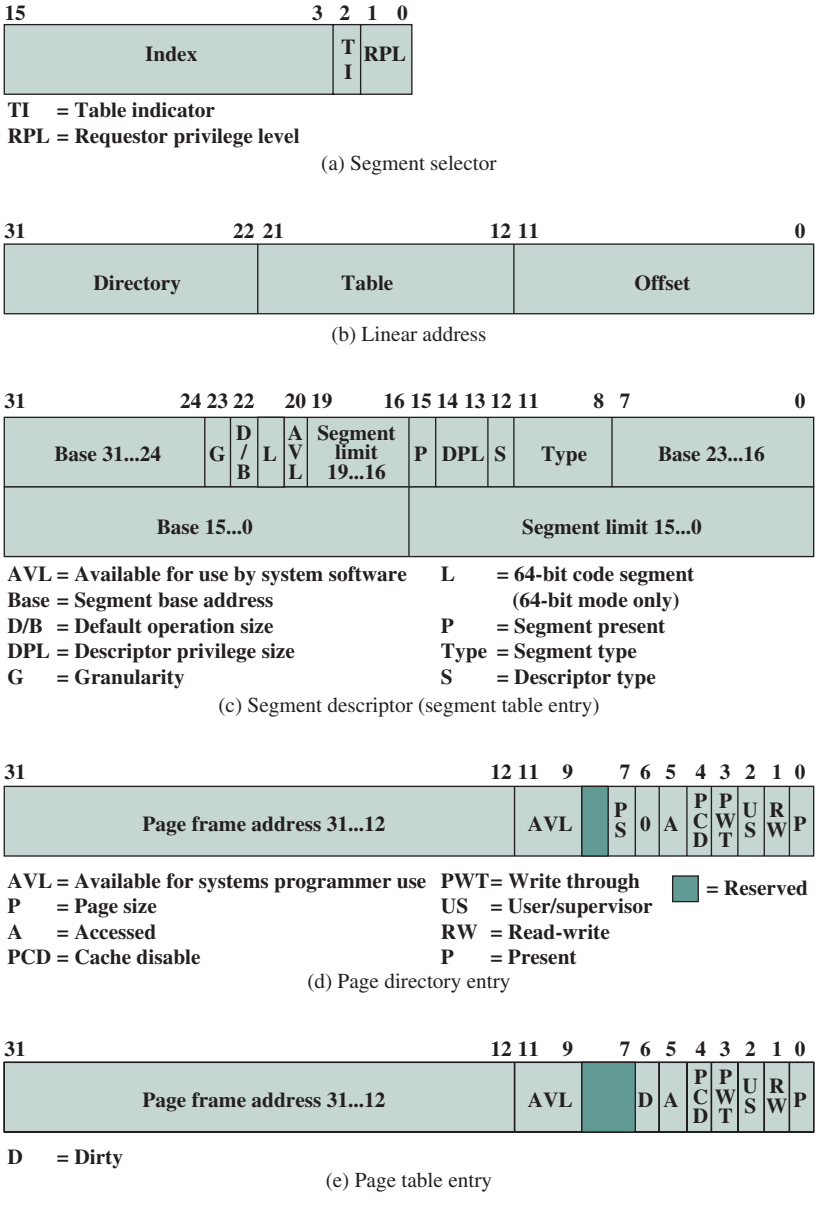


Figure 8.20 Intel x86 Memory Management Formats

- **Segment Number:** The number of the segment. This serves as an index into the segment table.
- **Requested Privilege Level (RPL):** The privilege level requested for this access.

Each entry in a segment table consists of 64 bits, as shown in Figure 8.20c. The fields are defined in Table 8.5.

Table 8.5 x86 Memory Management Parameters

Segment Descriptor (Segment Table Entry)

Base
Defines the starting address of the segment within the 4-Gbyte linear address space.

D/B bit
In a code segment, this is the D bit and indicates whether operands and addressing modes are 16 or 32 bits.

Descriptor Privilege Level (DPL)
Specifies the privilege level of the segment referred to by this segment descriptor.

Granularity bit (G)
Indicates whether the Limit field is to be interpreted in units by one byte or 4 Kbytes.

Limit
Defines the size of the segment. The processor interprets the limit field in one of two ways, depending on the granularity bit: in units of one byte, up to a segment size limit of 1 Mbyte, or in units of 4 Kbytes, up to a segment size limit of 4 Gbytes.

S bit
Determines whether a given segment is a system segment or a code or data segment.

Segment Present bit (P)
Used for nonpaged systems. It indicates whether the segment is present in main memory. For paged systems, this bit is always set to 1.

Type
Distinguishes between various kinds of segments and indicates the access attributes.

Page Directory Entry and Page Table Entry

Accessed bit (A)
This bit is set to 1 by the processor in both levels of page tables when a read or write operation to the corresponding page occurs.

Dirty bit (D)
This bit is set to 1 by the processor when a write operation to the corresponding page occurs.

Page Frame Address
Provides the physical address of the page in memory if the present bit is set. Since page frames are aligned on 4K boundaries, the bottom 12 bits are 0, and only the top 20 bits are included in the entry. In a page directory, the address is that of a page table.

Page Cache Disable bit (PCD)
Indicates whether data from page may be cached.

Page Size bit (PS)
Indicates whether page size is 4 Kbyte or 4 Mbyte.

Page Write Through bit (PWT)
Indicates whether write-through or write-back caching policy will be used for data in the corresponding page.

Present bit (P)
Indicates whether the page table or page is in main memory.

Read/Write bit (RW)
For user-level pages, indicates whether the page is read-only access or read/write access for user-level programs.

User/Supervisor bit (US)
Indicates whether the page is available only to the operating system (supervisor level) or is available to both operating system and applications (user level).

Paging

Segmentation is an optional feature and may be disabled. When segmentation is in use, addresses used in programs are virtual addresses and are converted into linear addresses, as just described. When segmentation is not in use, linear addresses are used in programs. In either case, the following step is to convert that linear address into a real 32-bit address.

To understand the structure of the linear address, you need to know that the x86 paging mechanism is actually a two-level table lookup operation. The first level is a page directory, which contains up to 1024 entries. This splits the 4-Gbyte linear memory space into 1024 page groups, each with its own page table, and each 4 Mbytes in length. Each page table contains up to 1024 entries; each entry corresponds to a single 4-Kbyte page. Memory management has the option of using one page directory for all processes, one page directory for each process, or some combination of the two. The page directory for the current task is always in main memory. Page tables may be in virtual memory.

Figure 8.20 shows the formats of entries in page directories and page tables, and the fields are defined in Table 8.5. Note that access control mechanisms can be provided on a page or page group basis.

The x86 also makes use of a translation lookaside buffer. The buffer can hold 32 page table entries. Each time that the page directory is changed, the buffer is cleared.

Figure 8.21 illustrates the combination of segmentation and paging mechanisms. For clarity, the translation lookaside buffer and memory cache mechanisms are not shown.

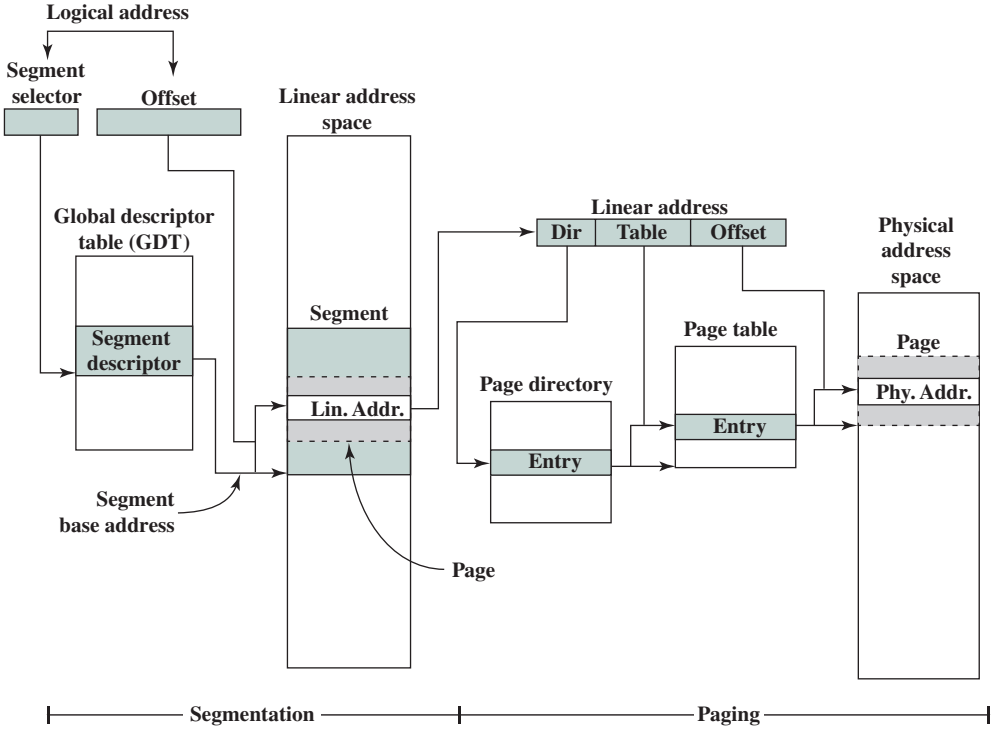


Figure 8.21 Intel x86 Memory Address Translation Mechanisms

Finally, the x86 includes a new extension not found on the earlier 80386 or 80486, the provision for two page sizes. If the PSE (page size extension) bit in control register 4 is set to 1, then the paging unit permits the OS programmer to define a page as either 4 Kbyte or 4 Mbyte in size.

When 4-Mbyte pages are used, there is only one level of table lookup for pages. When the hardware accesses the page directory, the page directory entry (Figure 8.20d) has the PS bit set to 1. In this case, bits 9 through 21 are ignored and bits 22 through 31 define the base address for a 4-Mbyte page in memory. Thus, there is a single page table.

The use of 4-Mbyte pages reduces the memory-management storage requirements for large main memories. With 4-Kbyte pages, a full 4-Gbyte main memory requires about 4 Mbytes of memory just for the page tables. With 4-Mbyte pages, a single table, 4 Kbytes in length, is sufficient for page memory management.

8.5 ARM MEMORY MANAGEMENT

ARM provides a versatile virtual memory system architecture that can be tailored to the needs of the embedded system designer.

Memory System Organization

Figure 8.22 provides an overview of the memory management hardware in the ARM for virtual memory. The virtual memory translation hardware uses one or two levels of tables for translation from virtual to physical addresses, as explained subsequently. The translation lookaside buffer (TLB) is a cache of recent page table entries. If an entry is available in the TLB, then the TLB directly sends a physical address to main memory for a read or write operation. As explained in Chapter 4, data is exchanged

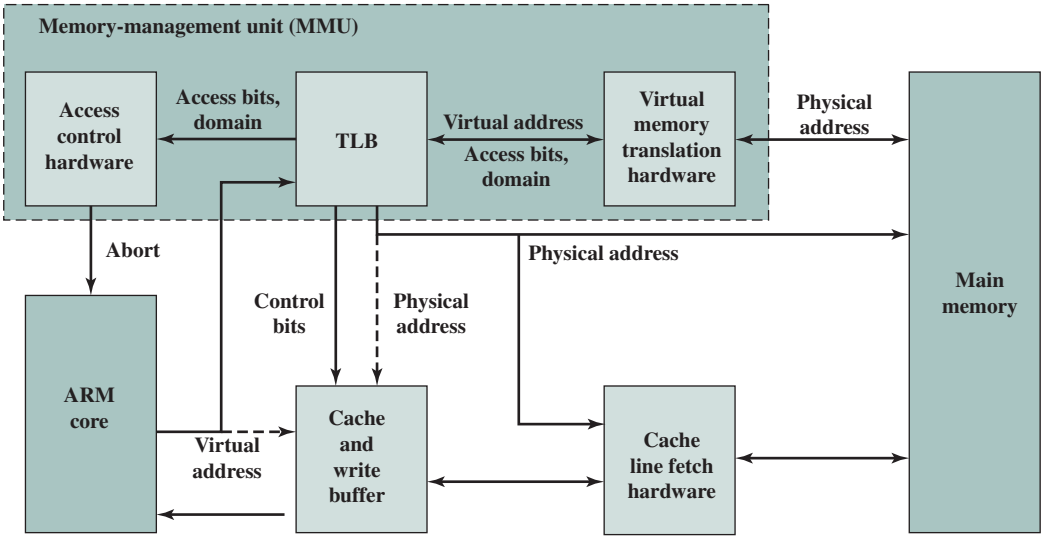


Figure 8.22 ARM Memory System Overview

between the processor and main memory via the cache. If a logical cache organization is used (Figure 4.7a), then the ARM supplies that address directly to the cache as well as supplying it to the TLB when a cache miss occurs. If a physical cache organization is used (Figure 4.7b), then the TLB must supply the physical address to the cache.

Entries in the translation tables also include access control bits, which determine whether a given process may access a given portion of memory. If access is denied, access control hardware supplies an abort signal to the ARM processor.

Virtual Memory Address Translation

The ARM supports memory access based on either sections or pages:

- **Supersections (optional):** Consist of 16-MB blocks of main memory.
- **Sections:** Consist of 1-MB blocks of main memory.
- **Large pages:** Consist of 64-kB blocks of main memory.
- **Small pages:** Consist of 4-kB blocks of main memory.

Sections and supersections are supported to allow mapping of a large region of memory while using only a single entry in the TLB. Additional access control mechanisms are extended within small pages to 1kB subpages, and within large pages to 16kB subpages. The translation table held in main memory has two levels:

- **Level 1 table:** Holds level 1 descriptors that contain the base address and translation properties for a Section and Supersection; and translation properties and pointers to a level 2 table for a large page or a small page.
- **Level 2 table:** Holds level 2 descriptors that contain the base address and translation properties for a Small page or a Large page. A level 2 table requires 1 kB of memory.

The memory-management unit (MMU) translates virtual addresses generated by the processor into physical addresses to access main memory, and also derives and checks the access permission. Translations occur as the result of a TLB miss, and start with a first-level fetch. A section-mapped access only requires a first-level fetch, whereas a page-mapped access also requires a second-level fetch.

Figure 8.23 shows the two-level address translation process for small pages. There is a single level 1 (L1) page table with 4K 32-bit entries. Each L1 entry points to a level 2 (L2) page table with 256 32-bit entries. Each of the L2 entry points to a 4-kB page in main memory. The 32-bit virtual address is interpreted as follows: The most significant 12 bits are an index into the L1 page table. The next 8 bits are an index into the relevant L2 page table. The least significant 12 bits index a byte in the relevant page in main memory.

A similar two-page lookup procedure is used for large pages. For sections and supersection, only the L1 page table lookup is required.

Memory-Management Formats

To get a better understanding of the ARM memory management scheme, we consider the key formats, as shown in Figure 8.24. The control bits shown in this figure are defined in Table 8.6.

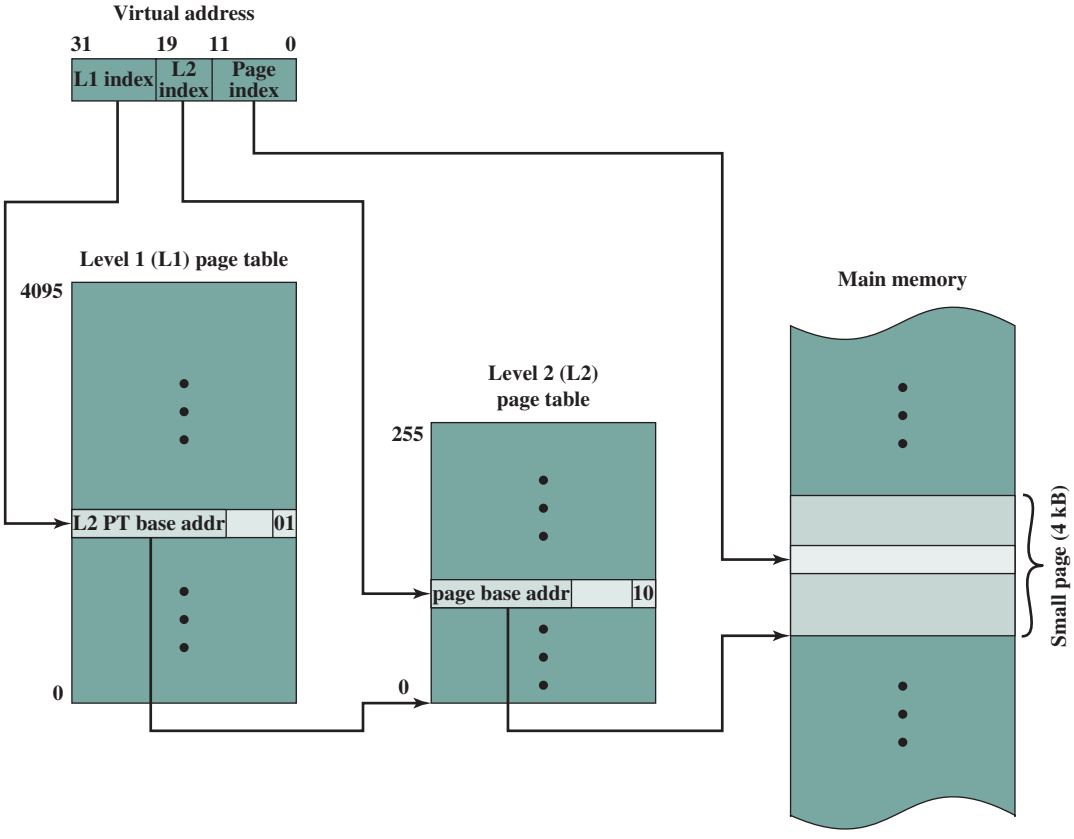


Figure 8.23 ARM Virtual Memory Address Translation for Small Pages

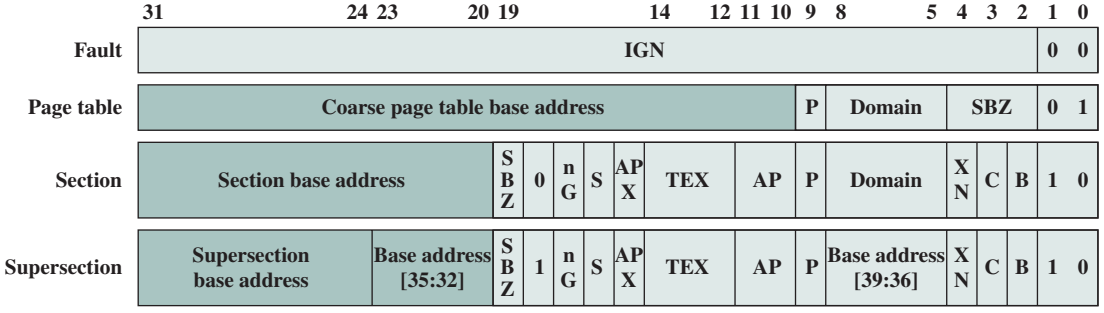
For the L1 table, each entry is a descriptor of how its associated 1-MB virtual address range is mapped. Each entry has one of four alternative formats:

- **Bits [1:0] = 00:** The associated virtual addresses are unmapped, and attempts to access them generate a translation fault.
- **Bits [1:0] = 01:** The entry gives the physical address of an L2 page table, which specifies how the associated virtual address range is mapped.
- **Bits [1:0] = 01: and bit 19 = 0:** The entry is a section descriptor for its associated virtual addresses.
- **Bits [1:0] = 01: and bit 19 = 1:** The entry is a supersection descriptor for its associated virtual addresses.

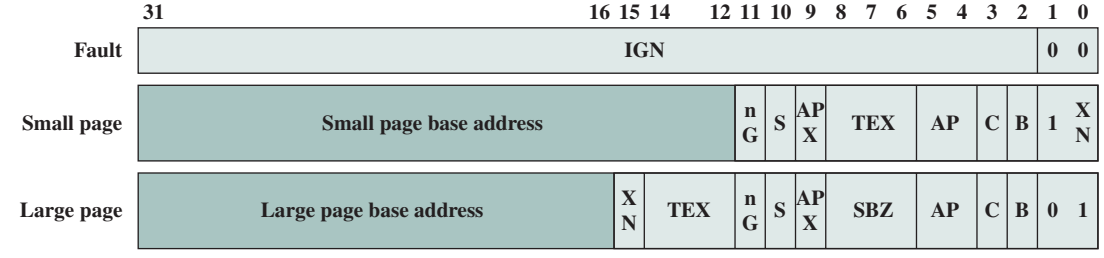
Entries with bits [1:0] = 11 are reserved.

For memory structured into pages, a two-level page table access is required. Bits [31:10] of the L1 page entry contain a pointer to a L2 page table. For small pages, the L2 entry contains a 20-bit pointer to the base address of a 4-kB page in main memory.

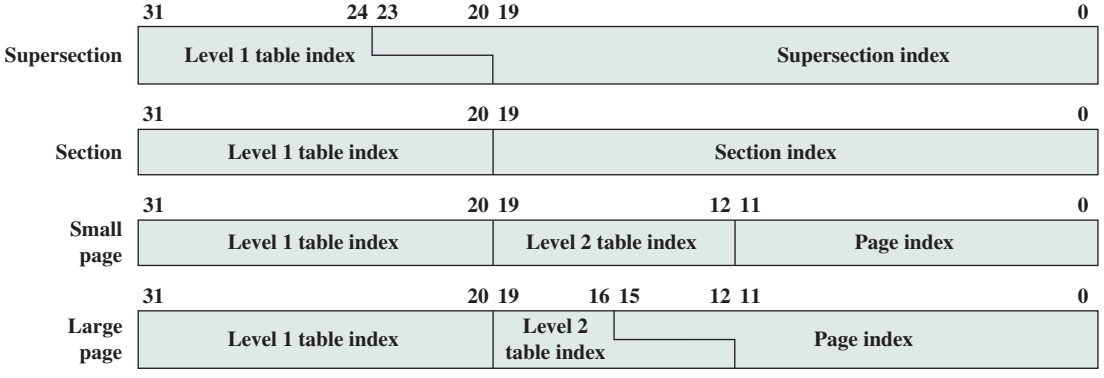
For large pages, the structure is more complex. As with virtual addresses for small pages, a virtual address for a large page structure includes a 12-bit index into



(a) Alternative first-level descriptor formats



(b) Alternative second-level descriptor formats



(c) Virtual memory address formats

Figure 8.24 ARM Memory-Management Formats

the level one table and an 8-bit index into the L2 table. For the 64-kB large pages, the page index portion of the virtual address must be 16 bits. To accommodate all of these bits in a 32-bit format, there is a 4-bit overlap between the page index field and the L2 table index field. ARM accommodates this overlap by requiring that each page table entry in a L2 page table that supports large pages be replicated 16 times. In effect, the size of the L2 page table is reduced from 256 entries to 16 entries, if all of the entries refer to large pages. However, a given L2 page can service a mixture of large and small pages, hence the need for the replication for large page entries.

Table 8.6 ARM Memory-Management Parameters

Access Permission (AP), Access Permission Extension (APX)
These bits control access to the corresponding memory region. If an access is made to an area of memory without the required permissions, a Permission Fault is raised.

Bufferable (B) bit
Determines, with the TEX bits, how the write buffer is used for cacheable memory.

Cacheable (C) bit
Determines whether this memory region can be mapped through the cache.

Domain
Collection of memory regions. Access control can be applied on the basis of domain.

not Global (nG)
Determines whether the translation should be marked as global (0), or process specific (1).

Shared (S)
Determines whether the translation is for not-shared (0), or shared (1) memory.

SBZ
Should be zero.

Type Extension (TEX)
These bits, together with the B and C bits, control accesses to the caches, how the write buffer is used, and if the memory region is shareable and therefore must be kept coherent.

Execute Never (XN)
Determines whether the region is executable (0) or not executable (1).

For memory structured into sections or supersections, a one-level page table access is required. For sections, bits [31:20] of the L1 entry contain a 12-bit pointer to the base of the 1-MB section in main memory.

For supersections, bits [31:24] of the L1 entry contain an 8-bit pointer to the base of the 16-MB section in main memory. As with large pages, a page table entry replication is required. In the case of supersections, the L1 table index portion of the virtual address overlaps by 4 bits with the supersection index portion of the virtual address Therefore, 16 identical L1 page table entries are required.

The range of physical address space can be expanded by up to eight additional address bits (bits [23:20] and [8:5]). The number of additional bits is implementation dependent. These additional bits can be interpreted as extending the size of physical memory by as much as a factor of $2^8 = 256$. Thus, physical memory may in fact be as much as 256 times as large as the memory space available to each individual process.

Access Control

The AP access control bits in each table entry control access to a region of memory by a given process. A region of memory can be designated as no access, read only, or read-write. Further, the region can be designated as privileged access only, reserved for use by the OS and not by applications.

ARM also employs the concept of a domain, which is a collection of sections and/or pages that have particular access permissions. The ARM architecture

supports 16 domains. The domain feature allows multiple processes to use the same translation tables while maintaining some protection from each other.

Each page table entry and TLB entry contains a field that specifies which domain the entry is in. A 2-bit field in the Domain Access Control Register controls access to each domain. Each field allows the access to an entire domain to be enabled and disabled very quickly, so that whole memory areas can be swapped in and out of virtual memory very efficiently. Two kinds of domain access are supported:

- **Clients:** Users of domains (execute programs and access data) that must observe the access permissions of the individual sections and/or pages that make up that domain.
- **Managers:** Control the behavior of the domain (the current sections and pages in the domain, and the domain access), and bypass the access permissions for table entries in that domain.

One program can be a client of some domains, and a manager of some other domains, and have no access to the remaining domains. This allows very flexible memory protection for programs that access different memory resources.

8.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

batch system	multitasking	resident monitor
demand paging	nucleus	segmentation
interactive operating system	operating system (OS)	short-term scheduling
interrupt	page table	swapping
job control language (JCL)	paging	thrashing
kernel	partitioning	time-sharing system
logical address	physical address	translation lookaside buffer (TLB)
long-term scheduling	privileged instruction	utility
medium-term scheduling	process	virtual memory
memory management	process control block	
memory protection	process state	
multiprogramming	real memory	

Review Questions

8.1 What is an operating system?
8.2 List and briefly define the key services provided by an OS.
8.3 List and briefly define the major types of OS scheduling.
8.4 What is the difference between a process and a program?
8.5 What is the purpose of swapping?
8.6 If a process may be dynamically assigned to different locations in main memory, what is the implication for the addressing mechanism?
8.7 Is it necessary for all of the pages of a process to be in main memory while the process is executing?

8.8 Must the pages of a process in main memory be contiguous?

8.9 Is it necessary for the pages of a process in main memory to be in sequential order?

8.10 What is the purpose of a translation lookaside buffer?

Problems

8.1 Suppose that we have a multiprogrammed computer in which each job has identical characteristics. In one computation period, T, for a job, half the time is spent in I/O and the other half in processor activity. Each job runs for a total of N periods. Assume that a simple round-robin priority is used, and that I/O operations can overlap with processor operation. Define the following quantities:

- Turnaround time = actual to complete a job.
- Throughput = average number of jobs completed per time period T.
- Processor utilization = percentage of time that the processor is active (not waiting).

Compute these quantities for one, two, and four simultaneous jobs, assuming that the period T is distributed in each of the following ways:

a. I/O first half, processor second half;

b. I/O first and fourth quarters, processor second and third quarters.

8.2 An I/O-bound program is one that, if run alone, would spend more time waiting for I/O than using the processor. A processor-bound program is the opposite. Suppose a short-term scheduling algorithm favors those programs that have used little processor time in the recent past. Explain why this algorithm favors I/O-bound programs and yet does not permanently deny processor time to processor-bound programs.

8.3 A program computes the row sums

$$C_i = \sum_{j=1}^{n} a_{ij}$$

of an array A that is 100 by 100. Assume that the computer uses demand paging with a page size of 1000 words, and that the amount of main memory allotted for data is five page frames. Is there any difference in the page fault rate if A were stored in virtual memory by rows or columns? Explain.

8.4 Consider a fixed partitioning scheme with equal-size partitions of 2^{16} bytes and a total main memory size of 2^{24} bytes. A process table is maintained that includes a pointer to a partition for each resident process. How many bits are required for the pointer?

8.5 Consider a dynamic partitioning scheme. Show that, on average, the memory contains half as many holes as segments.

8.6 Suppose the page table for the process currently executing on the processor looks like the following. All numbers are decimal, everything is numbered starting from zero, and all addresses are memory byte addresses. The page size is 1024 bytes.

Virtual page number	Valid bit	Reference bit	Modify bit	Page frame number
0	1	1	0	4
1	1	1	1	7
2	0	0	0	—
3	1	0	0	2
4	0	0	0	—
5	1	0	1	0

a. Describe exactly how, in general, a virtual address generated by the CPU is translated into a physical main memory address.
b. What physical address, if any, would each of the following virtual addresses correspond to? (Do not try to handle any page faults, if any.)
i. 1052
ii. 2221
iii. 5499

8.7 Give reasons that the page size in a virtual memory system should be neither very small nor very large.

8.8 A process references five pages, A, B, C, D, and E, in the following order:

A; B; C; D; A; B; E; A; B; C; D; E

Assume that the replacement algorithm is first-in-first-out and find the number of page transfers during this sequence of references starting with an empty main memory with three page frames. Repeat for four page frames.

8.9 The following sequence of virtual page numbers is encountered in the course of execution on a computer with virtual memory:

3 4 2 6 4 7 1 3 2 6 3 5 1 2 3

Assume that a least recently used page replacement policy is adopted. Plot a graph of page hit ratio (fraction of page references in which the page is in main memory) as a function of main-memory page capacity n for $1 \leq n \leq 8$. Assume that main memory is initially empty.

8.10 In the VAX computer, user page tables are located at virtual addresses in the system space. What is the advantage of having user page tables in virtual rather than main memory? What is the disadvantage?

8.11 Suppose the program statement

$$\textbf{for}\,(i = 1; i\,6 = n; i +)$$
$$\mathrm{a}[i] = \mathrm{b}[i] + \mathrm{c}[i];$$

is executed in a memory with page size of 1000 words. Let $n = 1000$. Using a machine that has a full range of register-to-register instructions and employs index registers, write a hypothetical program to implement the foregoing statement. Then show the sequence of page references during execution.

8.12 The IBM System/370 architecture uses a two-level memory structure and refers to the two levels as segments and pages, although the segmentation approach lacks many of the features described earlier in this chapter. For the basic 370 architecture, the page size may be either 2 Kbytes or 4 Kbytes, and the segment size is fixed at either 64 Kbytes or 1 Mbyte. For the 370/XA and 370/ESA architectures, the page size is 4 Kbytes and the segment size is 1 Mbyte. Which advantages of segmentation does this scheme lack? What is the benefit of segmentation for the 370?

8.13 Consider a computer system with both segmentation and paging. When a segment is in memory, some words are wasted on the last page. In addition, for a segment size s and a page size p, there are s/p page table entries. The smaller the page size, the less waste in the last page of the segment, but the larger the page table. What page size minimizes the total overhead?

8.14 A computer has a cache, main memory, and a disk used for virtual memory. If a referenced word is in the cache, 20 ns are required to access it. If it is in main memory but not in the cache, 60 ns are needed to load it into the cache, and then the reference is started again. If the word is not in main memory, 12 ms are required to fetch the word from disk, followed by 60 ns to copy it to the cache, and then the reference is started again. The cache hit ratio is 0.9 and the main-memory hit ratio is 0.6. What is the average time in ns required to access a referenced word on this system?

8.15 Assume a task is divided into four equal-sized segments and that the system builds an eight-entry page descriptor table for each segment. Thus, the system has a combination of segmentation and paging. Assume also that the page size is 2 Kbytes.

a. What is the maximum size of each segment?
b. What is the maximum logical address space for the task?
c. Assume that an element in physical location 00021ABC is accessed by this task. What is the format of the logical address that the task generates for it? What is the maximum physical address space for the system?

8.16 Assume a microprocessor capable of accessing up to 2^{32} bytes of physical main memory. It implements one segmented logical address space of maximum size 2^{31} bytes. Each instruction contains the whole two-part address. External memory management units (MMUs) are used, whose management scheme assigns contiguous blocks of physical memory of fixed size 2^{22} bytes to segments. The starting physical address of a segment is always divisible by 1024. Show the detailed interconnection of the external mapping mechanism that converts logical addresses to physical addresses using the appropriate number of MMUs, and show the detailed internal structure of an MMU (assuming that each MMU contains a 128-entry directly mapped segment descriptor cache) and how each MMU is selected.

8.17 Consider a paged logical address space (composed of 32 pages of 2 Kbytes each) mapped into a 1-Mbyte physical memory space.
a. What is the format of the processor's logical address?
b. What is the length and width of the page table (disregarding the "access rights" bits)?
c. What is the effect on the page table if the physical memory space is reduced by half?

8.18 In IBM's mainframe operating system, OS/390, one of the major modules in the kernel is the System Resource Manager (SRM). This module is responsible for the allocation of resources among address spaces (processes). The SRM gives OS/390 a degree of sophistication unique among operating systems. No other mainframe OS, and certainly no other type of OS, can match the functions performed by SRM. The concept of resource includes processor, real memory, and I/O channels. SRM accumulates statistics pertaining to utilization of processor, channel, and various key data structures. Its purpose is to provide optimum performance based on performance monitoring and analysis. The installation sets forth various performance objectives, and these serve as guidance to the SRM, which dynamically modifies installation and job performance characteristics based on system utilization. In turn, the SRM provides reports that enable the trained operator to refine the configuration and parameter settings to improve user service.

This problem concerns one example of SRM activity. Real memory is divided into equal-sized blocks called frames, of which there may be many thousands. Each frame can hold a block of virtual memory referred to as a page. SRM receives control approximately 20 times per second and inspects each and every page frame. If the page has not been referenced or changed, a counter is incremented by 1. Over time, SRM averages these numbers to determine the average number of seconds that a page frame in the system goes untouched. What might be the purpose of this and what action might SRM take?

8.19 For each of the ARM virtual address formats shown in Figure 8.24, show the physical address format.

8.20 Draw a figure similar to Figure 8.23 for ARM virtual memory translation when main memory is divided into sections.

PART THREE ARITHMETIC AND LOGIC

CHAPTER 9

NUMBER SYSTEMS

9.1 **The Decimal System**

9.2 **Positional Number Systems**

9.3 **The Binary System**

9.4 **Converting Between Binary and Decimal**
- Integers
- Fractions

9.5 **Hexadecimal Notation**

9.6 **Key Terms and Problems**

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Understand the basic concepts and terminology of **positional number systems**.
- Explain the techniques for converting between **decimal** and **binary** for both integers and fractions.
- Explain the rationale for using **hexadecimal notation**.

9.1 THE DECIMAL SYSTEM

In everyday life we use a system based on decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) to represent numbers, and refer to the system as the decimal system. Consider what the number 83 means. It means eight tens plus three:

$$83 = (8 \times 10) + 3$$

The number 4728 means four thousands, seven hundreds, two tens, plus eight:

$$4728 = (4 \times 1000) + (7 \times 100) + (2 \times 10) + 8$$

The decimal system is said to have a **base**, or **radix**, of 10. This means that each digit in the number is multiplied by 10 raised to a power corresponding to that digit's position:

$$83 = (8 \times 10^1) + (3 \times 10^0)$$
$$4728 = (4 \times 10^3) + (7 \times 10^2) + (2 \times 10^1) + (8 \times 10^0)$$

The same principle holds for decimal fractions, but negative powers of 10 are used. Thus, the decimal fraction 0.256 stands for 2 tenths plus 5 hundredths plus 6 thousandths:

$$0.256 = (2 \times 10^{-1}) + (5 \times 10^{-2}) + (6 \times 10^{-3})$$

A number with both an integer and fractional part has digits raised to both positive and negative powers of 10:

$$442.256 = (4 \times 10^2) + (4 + 10^1) + (2 \times 10^0) + (2 \times 10^{-1}) + (5 \times 10^{-2}) + (6 \times 10^{-3})$$

In any number, the leftmost digit is referred to as the **most significant digit**, because it carries the highest value. The rightmost digit is called the **least significant digit**. In the preceding decimal number, the 4 on the left is the most significant digit and the 6 on the right is the least significant digit.

Table 9.1 shows the relationship between each digit position and the value assigned to that position. Each position is weighted 10 times the value of the position to the right and one-tenth the value of the position to the left. Thus, positions represent successive powers of 10. If we number the positions as indicated in Table 9.1, then position i is weighted by the value 10^i.

Table 9.1 Positional Interpretation of a Decimal Number

4	7	2	2	5	6
100s	10s	1s	tenths	hundredths	thousandths
10^2	10^1	10^0	10^{-1}	10^{-2}	10^{-3}
position 2	position 1	position 0	position –1	position –2	position –3

In general, for the decimal representation of $X = \{\ldots d_2 d_1 d_0 . d_{-1} d_{-2} d_{-3} \ldots\}$, the value of X is

$$X = \sum_i (d_i \times 10^i) \tag{9.1}$$

One other observation is worth making. Consider the number 509 and ask how many tens are in the number. Because there is a 0 in the tens position, you might be tempted to say there are no tens. But there are in fact 50 tens. What the 0 in the tens position means is that there are no tens left over that cannot be lumped into the hundreds, or thousands, and so on. Therefore, because each position holds only the leftover numbers that cannot be lumped into higher positions, each digit position needs to have a value of no greater than nine. Nine is the maximum value that a position can hold before it flips over into the next higher position.

9.2 POSITIONAL NUMBER SYSTEMS

In a positional number system, each number is represented by a string of digits in which each digit position i has an associated weight r^i, where r is the radix, or base, of the number system. The general form of a number in such a system with radix r is

$$(\ldots a_3 a_2 a_1 a_0 . a_{-1} a_{-2} a_{-3} \ldots)_r$$

where the value of any digit a_i is an integer in the range $0 \leq a_i < r$. The dot between a_0 and a_{-1} is called the **radix point**. The number is defined to have the value

$$\ldots + a_3 r^3 + a_2 r^2 + a_1 r^1 + a_0 r^0 + a_{-1} r^{-1} + a_{-2} r^{-2} + a_{-3} r^{-3} + \ldots$$

$$= \sum_i (a_i \times b^i) \tag{9.2}$$

The decimal system, then, is a special case of a positional number system with radix 10 and with digits in the range 0 through 9.

As an example of another positional system, consider the system with base 7. Table 9.2 shows the weighting value for positions −1 through 4. In each position, the digit value ranges from 0 through 6.

Table 9.2 Positional Interpretation of a Number in Base 7

Position	4	3	2	1	0	−1
Value in Exponential Form	7^4	7^3	7^2	7^1	7^0	7^{-1}
Decimal Value	2401	343	49	7	1	1/7

9.3 THE BINARY SYSTEM

In the decimal system, 10 different digits are used to represent numbers with a base of 10. In the binary system, we have only two digits, 1 and 0. Thus, numbers in the binary system are represented to base 2.

To avoid confusion, we will sometimes put a subscript on a number to indicate its base. For example, 83_{10} and 4728_{10} are numbers represented in decimal notation or, more briefly, decimal numbers. The digits 1 and 0 in binary notation have the same meaning as in decimal notation:

$$0_2 = 0_{10}$$
$$1_2 = 1_{10}$$

To represent larger numbers, as with decimal notation, each digit in a binary number has a value depending on its position:

$$10_2 = (1 \times 2^1) + (0 \times 2^0) = 2_{10}$$
$$11_2 = (1 \times 2^1) + (1 \times 2^0) = 3_{10}$$
$$100_2 = (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) = 4_{10}$$

and so on. Again, fractional values are represented with negative powers of the radix:

$$1001.101 = 2^3 + 2^0 + 2^{-1} + 2^{-3} = 9.625_{10}$$

In general, for the binary representation of $Y = \{\ldots b_2 b_1 b_0 . b_{-1} b_{-2} b_{-3} \ldots\}$, the value of Y is

$$Y = \sum_i (b_i \times 2^i) \tag{9.3}$$

9.4 CONVERTING BETWEEN BINARY AND DECIMAL

It is a simple matter to convert a number from binary notation to decimal notation. In fact, we showed several examples in the previous subsection. All that is required is to multiply each binary digit by the appropriate power of 2 and add the results.

To convert from decimal to binary, the integer and fractional parts are handled separately.

Integers

For the integer part, recall that in binary notation, an integer represented by

$$b_{m-1} b_{m-2} \ldots b_2 b_1 b_0 \qquad b_i = 0 \text{ or } 1$$

has the value

$$(b_{m-1} \times 2^{m-1}) + (b_{m-2} \times 2^{m-2}) + \ldots + (b_1 \times 2^1) + b_0$$

Suppose it is required to convert a decimal integer N into binary form. If we divide N by 2, in the decimal system, and obtain a quotient N_1 and a remainder R_0, we may write

$$N = 2 \times N_1 + R_0 \qquad R_0 = 0 \text{ or } 1$$

Next, we divide the quotient N_1 by 2. Assume that the new quotient is N_2 and the new remainder R_1. Then

$$N_1 = 2 \times N_2 + R_1 \qquad R_1 = 0 \text{ or } 1$$

so that

$$N = 2(2N_2 + R_1) + R_0 = (N_2 \times 2^2) + (R_1 \times 2^1) + R_0$$

If next

$$N_2 = 2N_3 + R_2$$

we have

$$N = (N_3 \times 2^3) + (R_2 \times 2^2) + (R_1 \times 2^1) + R_0$$

Because $N > N_1 > N_2 \ldots$, continuing this sequence will eventually produce a quotient $N_{m-1} = 1$ (except for the decimal integers 0 and 1, whose binary equivalents are 0 and 1, respectively) and a remainder R_{m-2}, which is 0 or 1. Then

$$N = (1 \times 2^{m-1}) + (R_{m-2} \times 2^{m-2}) + \ \ldots \ + (R_2 \times 2^2) + (R_1 \times 2^1) + R_0$$

which is the binary form of N. Hence, we convert from base 10 to base 2 by repeated divisions by 2. The remainders and the final quotient, 1, give us, in order of increasing significance, the binary digits of N. Figure 9.1 shows two examples.

Fractions

For the fractional part, recall that in binary notation, a number with a value between 0 and 1 is represented by

$$0.b_{-1}b_{-2}b_{-3} \ldots \qquad b_i = 0 \text{ or } 1$$

and has the value

$$(b_{-1} \times 2^{-1}) + (b_{-2} \times 2^{-2}) + (b_{-3} \times 2^{-3}) \ldots$$

This can be rewritten as

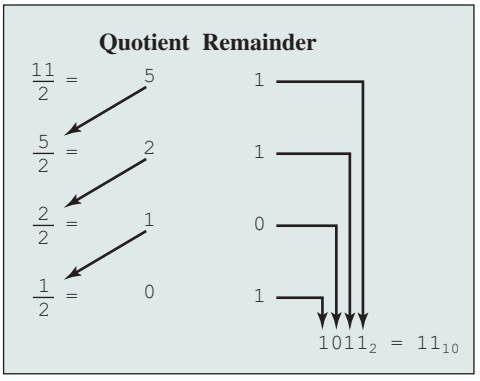
$$2^{-1} \times (b_{-1} + 2^{-1} \times (b_{-2} + 2^{-1} \times (b_{-3} + \ \ldots) \ldots))$$

This expression suggests a technique for conversion. Suppose we want to convert the number $F (0 < F < 1)$ from decimal to binary notation. We know that F can be expressed in the form

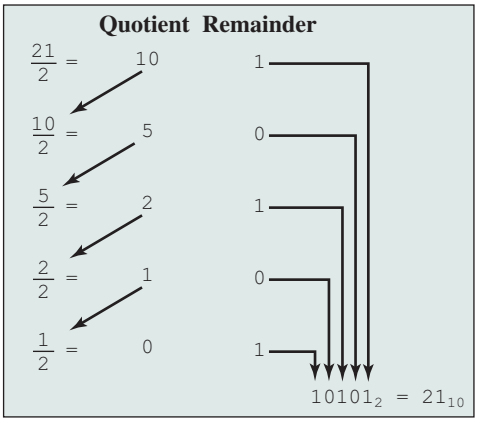
$$F = 2^{-1} \times (b_{-1} + 2^{-1} \times (b_{-2} + 2^{-1} \times (b_{-3} + \ \ldots) \ \ldots))$$

If we multiply F by 2, we obtain,

$$2 \times F = b_{-1} + 2^{-1} \times (b_{-2} + 2^{-1} \times (b_{-3} + \ \ldots) \ldots)$$



(a) 11_{10}



(b) 21_{10}

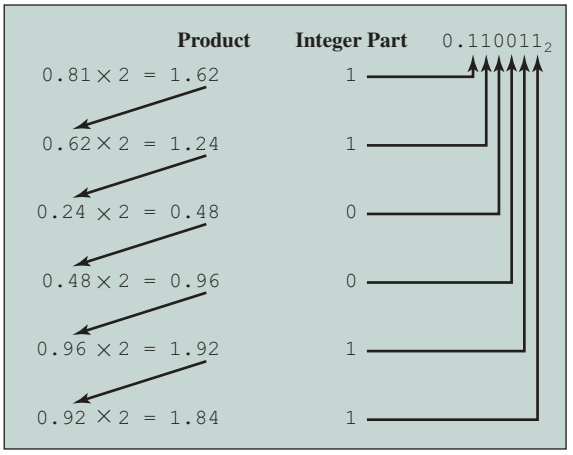
Figure 9.1 Examples of Converting from Decimal Notation to Binary Notation for Integers

From this equation, we see that the integer part of $(2 \times F)$, which must be either 0 or 1 because $0 < F < 1$, is simply b_{-1}. So we can say $(2 \times F) = b_{-1} + F_1$, where $0 < F_1 < 1$ and where

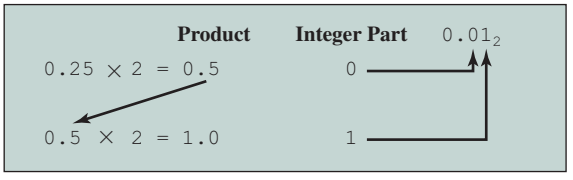
$$F_1 = 2^{-1} \times (b_{-2} + 2^{-1} \times (b_{-3} + 2^{-1} \times (b_{-4} + \ \ldots) \ldots))$$

To find b_{-2}, we repeat the process. Therefore, the conversion algorithm involves repeated multiplication by 2. At each step, the fractional part of the number from the previous step is multiplied by 2. The digit to the left of the decimal point in the product will be 0 or 1 and contributes to the binary representation, starting with the most significant digit. The fractional part of the product is used as the multiplicand in the next step. Figure 9.2 shows two examples.

This process is not necessarily exact; that is, a decimal fraction with a finite number of digits may require a binary fraction with an infinite number of digits. In such cases, the conversion algorithm is usually halted after a prespecified number of steps, depending on the desired accuracy.



(a) $0.81_{10} = 0.110011_2$ (approximately)



(b) $0.25_{10} = 0.01_2$ (exactly)

Figure 9.2 Examples of Converting from Decimal Notation to Binary Notation for Fractions

9.5 HEXADECIMAL NOTATION

Because of the inherent binary nature of digital computer components, all forms of data within computers are represented by various binary codes. However, no matter how convenient the binary system is for computers, it is exceedingly cumbersome for human beings. Consequently, most computer professionals who must spend time working with the actual raw data in the computer prefer a more compact notation.

What notation to use? One possibility is the decimal notation. This is certainly more compact than binary notation, but it is awkward because of the tediousness of converting between base 2 and base 10.

Instead, a notation known as hexadecimal has been adopted. Binary digits are grouped into sets of four bits, called a **nibble**. Each possible combination of four binary digits is given a symbol, as follows:

0000 = 0	0100 = 4	1000 = 8	1100 = C
0001 = 1	0101 = 5	1001 = 9	1101 = D
0010 = 2	0110 = 6	1010 = A	1110 = E
0011 = 3	0111 = 7	1011 = B	1111 = F

Because 16 symbols are used, the notation is called **hexadecimal**, and the 16 symbols are the **hexadecimal digits**.

A sequence of hexadecimal digits can be thought of as representing an integer in base 16 (Table 9.3). Thus,

$$\begin{aligned} 2C_{16} &= (2_{16} \times 16^1) + (C_{16} \times 16^0) \\ &= (2_{10} \times 16^1) + (12_{10} \times 16^0) = 44 \end{aligned}$$

Thus, viewing hexadecimal numbers as numbers in the positional number system with base 16, we have

$$Z = \sum_i (h_i \times 16^i) \tag{9.4}$$

where 16 is the base and each hexadecimal digit h_i is in the decimal range $0 \leq h_i < 15$, equivalent to the hexadecimal range $0 \leq h_i \leq \text{F}$.

Table 9.3 Decimal, Binary, and Hexadecimal

Decimal (base 10)	Binary (base 2)	Hexadecimal (base 16)
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	0001 0000	10
17	0001 0001	11
18	0001 0010	12
31	0001 1111	1F
100	0110 0100	64
255	1111 1111	FF
256	0001 0000 0000	100

Hexadecimal notation is not only used for representing integers but also used as a concise notation for representing any sequence of binary digits, whether they represent text, numbers, or some other type of data. The reasons for using hexadecimal notation are as follows:

1. It is more compact than binary notation.
2. In most computers, binary data occupy some multiple of 4 bits, and hence some multiple of a single hexadecimal digit.
3. It is extremely easy to convert between binary and hexadecimal notation.

As an example of the last point, consider the binary string 110111100001. This is equivalent to

$$\begin{array}{ccc} 1101 & 1110 & 0001 = \mathrm{DE1}_{16} \\ \mathrm{D} & \mathrm{E} & 1 \end{array}$$

This process is performed so naturally that an experienced programmer can mentally convert visual representations of binary data to their hexadecimal equivalent without written effort.

9.6 KEY TERMS AND PROBLEMS

Key Terms

base	hexadecimal	nibble
binary	integer	positional number system
decimal	least significant digit	radix
fraction	most significant digit	radix point

Problems

9.1 Count from 1 to 20_{10} in the following bases:
a. 8 **b.** 6 **c.** 5 **d.** 3

9.2 Order the numbers $(1.1)_2$, $(1.4)_{10}$, and $(1.5)_{16}$ from smallest to largest.

9.3 Perform the indicated base conversions:
a. 54_8 to base 5 **b.** 312_4 to base 7 **c.** 520_6 to base 7 **d.** 12212_3 to base 9

9.4 What generalizations can you draw about converting a number from one base to a power of that base; e.g., from base 3 to base 9 (3^2) or from base 2 to base 4 (2^2) or base 8 (2^3)?

9.5 Convert the following binary numbers to their decimal equivalents:
a. 001100 **b.** 000011 **c.** 011100 **d.** 111100 **e.** 101010

9.6 Convert the following binary numbers to their decimal equivalents:
a. 11100.011 **b.** 110011.10011 **c.** 1010101010.1

9.7 Convert the following decimal numbers to their binary equivalents:
a. 64 **b.** 100 **c.** 111 **d.** 145 **e.** 255

9.8 Convert the following decimal numbers to their binary equivalents:
a. 34.75 **b.** 25.25 **c.** 27.1875

9.9 Prove that every real number with a terminating binary representation (finite number of digits to the right of the binary point) also has a terminating decimal representation (finite number of digits to the right of the decimal point).

9.10 Express the following octal numbers (number with radix 8) in hexadecimal notation:

a. 12 **b.** 5655 **c.** 2550276 **d.** 76545336 **e.** 3726755

9.11 Convert the following hexadecimal numbers to their decimal equivalents:

a. C **b.** 9F **c.** D52 **d.** 67E **e.** ABCD

9.12 Convert the following hexadecimal numbers to their decimal equivalents:

a. F.4 **b.** D3.E **c.** 1111.1 **d.** 888.8 **e.** EBA.C

9.13 Convert the following decimal numbers to their hexadecimal equivalents:

a. 16 **b.** 80 **c.** 2560 **d.** 3000 **e.** 62,500

9.14 Convert the following decimal numbers to their hexadecimal equivalents:

a. 204.125 **b.** 255.875 **c.** 631.25 **d.** 10000.00390625

9.15 Convert the following hexadecimal numbers to their binary equivalents:

a. E **b.** 1C **c.** A64 **d.** 1F.C **e.** 239.4

9.16 Convert the following binary numbers to their hexadecimal equivalents:

a. 1001.1111 **b.** 110101.011001 **c.** 10100111.111011

CHAPTER 10

COMPUTER ARITHMETIC

10.1 The Arithmetic and Logic Unit

10.2 Integer Representation
- Sign-Magnitude Representation
- Twos Complement Representation
- Range Extension
- Fixed-Point Representation

10.3 Integer Arithmetic
- Negation
- Addition and Subtraction
- Multiplication
- Division

10.4 Floating-Point Representation
- Principles
- IEEE Standard for Binary Floating-Point Representation

10.5 Floating-Point Arithmetic
- Addition and Subtraction
- Multiplication and Division
- Precision Considerations
- IEEE Standard for Binary Floating-Point Arithmetic

10.6 Key Terms, Review Questions, and Problems

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Understand the distinction between the way in which numbers are represented (the binary format) and the algorithms used for the basic arithmetic operations.
- Explain **twos complement representation**.
- Present an overview of the techniques for doing basic arithmetic operation in two complement notation.
- Understand the use of significand, base, and exponent in the representation of **floating-point numbers**.
- Present an overview of the IEEE 754 standard for floating-point representation.
- Understand some of the key concepts related to floating-point arithmetic, including guard bits, rounding, subnormal numbers, underflow and overflow.

We begin our examination of the processor with an overview of the arithmetic and logic unit (ALU). The chapter then focuses on the most complex aspect of the ALU, computer arithmetic. The implementations of simple logic and arithmetic functions in digital logic are described in Chapter 11, and logic functions that are part of the ALU are described in Chapter 12.

Computer arithmetic is commonly performed on two very different types of numbers: integer and floating point. In both cases, the representation chosen is a crucial design issue and is treated first, followed by a discussion of arithmetic operations.

This chapter includes a number of examples, each of which is highlighted in a shaded box.

10.1 THE ARITHMETIC AND LOGIC UNIT

The ALU is that part of the computer that actually performs arithmetic and logical operations on data. All of the other elements of the computer system—control unit, registers, memory, I/O—are there mainly to bring data into the ALU for it to process and then to take the results back out. We have, in a sense, reached the core or essence of a computer when we consider the ALU.

An ALU and indeed, all electronic components in the computer, are based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations.

Figure 10.1 indicates, in general terms, how the ALU is interconnected with the rest of the processor. Operands for arithmetic and logic operations are presented to the ALU in registers, and the results of an operation are stored in registers. These registers are temporary storage locations within the processor that are connected by signal paths to the ALU (e.g., see Figure 2.3). The ALU may also set flags as the result of an operation. For example, an overflow flag is set to 1 if the result of a computation exceeds the length of the register into which it is to be stored.

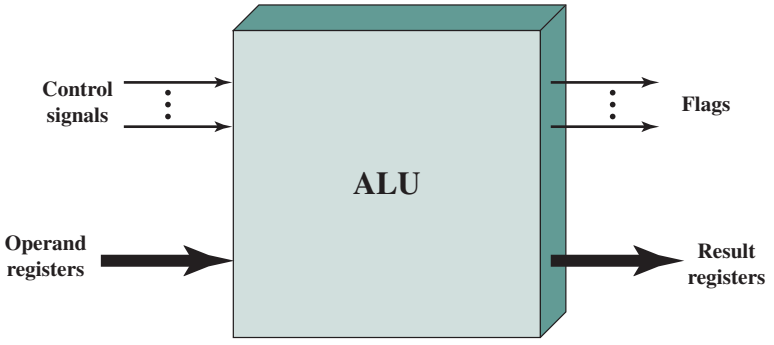


Figure 10.1 ALU Inputs and Outputs

The flag values are also stored in registers within the processor. The processor provides signals that control the operation of the ALU and the movement of the data into and out of the ALU.

10.2 INTEGER REPRESENTATION

In the binary number system,[1] arbitrary numbers can be represented with just the digits zero and one, the minus sign (for negative numbers), and the period, or **radix point** (for numbers with a fractional component).

$$-1101.0101_2 = -13.3125_{10}$$

For purposes of computer storage and processing, however, we do not have the benefit of special symbols for the minus sign and radix point. Only binary digits (0 and 1) may be used to represent numbers. If we are limited to nonnegative integers, the representation is straightforward.

An 8-bit word can represent the numbers from 0 to 255, such as

$$\begin{aligned} 00000000 &= \quad 0 \\ 00000001 &= \quad 1 \\ 00101001 &= \quad 41 \\ 10000000 &= 128 \\ 11111111 &= 255 \end{aligned}$$

In general, if an n-bit sequence of binary digits $a_{n-1}a_{n-2} \ldots a_1a_0$ is interpreted as an unsigned integer A, its value is

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

[1]See Chapter 9 for a basic refresher on number systems (decimal, binary, hexadecimal).

Sign-Magnitude Representation

There are several alternative conventions used to represent negative as well as positive integers, all of which involve treating the most significant (leftmost) bit in the word as a sign bit. If the sign bit is 0, the number is positive; if the sign bit is 1, the number is negative.

The simplest form of representation that employs a sign bit is the sign-magnitude representation. In an n-bit word, the rightmost $n - 1$ bits hold the magnitude of the integer.

$$\begin{aligned} +18 &= 00010010 \\ -18 &= 10010010 \quad \text{(sign magnitude)} \end{aligned}$$

The general case can be expressed as follows:

Sign Magnitude

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases} \tag{10.1}$$

There are several drawbacks to sign-magnitude representation. One is that addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation. This should become clear in the discussion in Section 10.3. Another drawback is that there are two representations of 0:

$$\begin{aligned} +0_{10} &= 00000000 \\ -0_{10} &= 10000000 \quad \text{(sign magnitude)} \end{aligned}$$

This is inconvenient because it is slightly more difficult to test for 0 (an operation performed frequently on computers) than if there were a single representation.

Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU. Instead, the most common scheme is twos complement representation.[2]

Twos Complement Representation

Like sign magnitude, twos complement representation uses the most significant bit as a sign bit, making it easy to test whether an integer is positive or negative. It differs from the use of the sign-magnitude representation in the way that the other bits are interpreted. Table 10.1 highlights key characteristics of twos complement representation and arithmetic, which are elaborated in this section and the next.

Most treatments of twos complement representation focus on the rules for producing negative numbers, with no formal proof that the scheme is valid. Instead,

[2]In the literature, the terms *two's complement* or *2's complement* are often used. Here we follow the practice used in standards documents and omit the apostrophe (e.g., IEEE Std 100-1992, *The New IEEE Standard Dictionary of Electrical and Electronics Terms*).

Table 10.1 Characteristics of Twos Complement Representation and Arithmetic

Range	-2^{n-1} through $2^{n-1} - 1$
Number of Representations of Zero	One
Negation	Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
Expansion of Bit Length	Add additional bit positions to the left and fill in with the value of the original sign bit.
Overflow Rule	If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.
Subtraction Rule	To subtract B from A, take the twos complement of B and add it to A.

our presentation of twos complement integers in this section and in Section 10.3 is based on [DATT93], which suggests that twos complement representation is best understood by defining it in terms of a weighted sum of bits, as we did previously for unsigned and sign-magnitude representations. The advantage of this treatment is that it does not leave any lingering doubt that the rules for arithmetic operations in twos complement notation may not work for some special cases.

Consider an n-bit integer, A, in twos complement representation. If A is positive, then the sign bit, a_{n-1}, is zero. The remaining bits represent the magnitude of the number in the same fashion as for sign magnitude:

$$A = \sum_{i=0}^{n-2} 2^i a_i \qquad \text{for } A \geq 0$$

The number zero is identified as positive and therefore has a 0 sign bit and a magnitude of all 0s. We can see that the range of positive integers that may be represented is from 0 (all of the magnitude bits are 0) through $2^{n-1} - 1$ (all of the magnitude bits are 1). Any larger number would require more bits.

Now, for a negative number $A(A < 0)$, the sign bit, a_{n-1}, is one. The remaining $n - 1$ bits can take on any one of 2^{n-1} values. Therefore, the range of negative integers that can be represented is from -1 to -2^{n-1}. We would like to assign the bit values to negative integers in such a way that arithmetic can be handled in a straightforward fashion, similar to unsigned integer arithmetic. In unsigned integer representation, to compute the value of an integer from the bit representation, the weight of the most significant bit is $+2^{n-1}$. For a representation with a sign bit, it turns out that the desired arithmetic properties are achieved, as we will see in Section 10.3, if the weight of the most significant bit is -2^{n-1}. This is the convention used in twos complement representation, yielding the following expression for negative numbers:

Twos Complement

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \qquad \textbf{(10.2)}$$

Equation (10.2) defines the twos complement representation for both positive and negative numbers. For $a_{n-1} = 0$, the term $-2^{n-1}a_{n-1} = 0$ and the equation defines

Table 10.2 Alternative Representations for 4-Bit Integers

Decimal Representation	Sign-Magnitude Representation	Twos Complement Representation	Biased Representation
+8	—	—	1111
+7	0111	0111	1110
+6	0110	0110	1101
+5	0101	0101	1100
+4	0100	0100	1011
+3	0011	0011	1010
+2	0010	0010	1001
+1	0001	0001	1000
+0	0000	0000	0111
−0	1000	—	—
−1	1001	1111	0110
−2	1010	1110	0101
−3	1011	1101	0100
−4	1100	1100	0011
−5	1101	1011	0010
−6	1110	1010	0001
−7	1111	1001	0000
−8	—	1000	—

a nonnegative integer. When $a_{n-1} = 1$, the term 2^{n-1} is subtracted from the summation term, yielding a negative integer.

Table 10.2 compares the sign-magnitude and twos complement representations for 4-bit integers. Although twos complement is an awkward representation from the human point of view, we will see that it facilitates the most important arithmetic operations, addition and subtraction. For this reason, it is almost universally used as the processor representation for integers.

A useful illustration of the nature of twos complement representation is a value box, in which the value on the far right in the box is 1 (2^0) and each succeeding position to the left is double in value, until the leftmost position, which is negated. As you can see in Figure 10.2a, the most negative twos complement number that can be represented is -2^{n-1}; if any of the bits other than the sign bit is one, it adds a positive amount to the number. Also, it is clear that a negative number must have a 1 at its leftmost position and a positive number must have a 0 in that position. Thus, the largest positive number is a 0 followed by all 1s, which equals $2^{n-1} - 1$.

The rest of Figure 10.2 illustrates the use of the value box to convert from twos complement to decimal and from decimal to twos complement.

Range Extension

It is sometimes desirable to take an n-bit integer and store it in m bits, where $m > n$. This expansion of bit length is referred to as **range extension**, because the range of numbers that can be expressed is extended by increasing the bit length.

−128	64	32	16	8	4	2	1

(a) An eight-position twos complement value box

−128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

−128 +2 +1 = −125

(b) Convert binary 10000011 to decimal

−128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0

−120 = −128 +8

(c) Convert decimal −120 to binary

Figure 10.2 Use of a Value Box for Conversion between Twos Complement Binary and Decimal

In sign-magnitude notation, this is easily accomplished: simply move the sign bit to the new leftmost position and fill in with zeros.

+18	=	00010010	(sign magnitude, 8 bits)
+18	=	0000000000010010	(sign magnitude, 16 bits)
−18	=	10010010	(sign magnitude, 8 bits)
−18	=	1000000000010010	(sign magnitude, 16 bits)

This procedure will not work for twos complement negative integers. Using the same example,

+18	=	00010010	(twos complement, 8 bits)
+18	=	0000000000010010	(twos complement, 16 bits)
−18	=	11101110	(twos complement, 8 bits)
−32,658	=	1000000001101110	(twos complement, 16 bits)

The next to last line is easily seen using the value box of Figure 10.2. The last line can be verified using Equation (10.2) or a 16-bit value box.

Instead, the rule for twos complement integers is to move the sign bit to the new leftmost position and fill in with copies of the sign bit. For positive numbers, fill in with zeros, and for negative numbers, fill in with ones. This is called sign extension.

−18	=	11101110	(twos complement, 8 bits)
−18	=	1111111111101110	(twos complement, 16 bits)

To see why this rule works, let us again consider an n-bit sequence of binary digits $a_{n-1}a_{n-2} \ldots a_1a_0$ interpreted as a twos complement integer A, so that its value is

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

If A is a positive number, the rule clearly works. Now, if A is negative and we want to construct an m-bit representation, with $m > n$. Then

$$A = -2^{m-1}a_{m-1} + \sum_{i=0}^{m-2} 2^i a_i$$

The two values must be equal:

$$-2^{m-1} + \sum_{i=0}^{m-2} 2^i a_i = -2^{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

$$-2^{m-1} + \sum_{i=n-1}^{m-2} 2^i a_i = -2^{n-1}$$

$$-2^{n-1} + \sum_{i=n-1}^{m-2} 2^i a_i = 2^{m-1}$$

$$1 + \sum_{i=0}^{n-2} 2^i + \sum_{i=n-1}^{m-2} 2^i a_i = 1 + \sum_{i=0}^{m-2} 2^i$$

$$\sum_{i=n-1}^{m-2} 2^i a_i = \sum_{i=n-1}^{m-2} 2^i$$

$$\Rightarrow a_{m-2} = \ldots = a_{n-2} = a_{n-2} = 1$$

In going from the first to the second equation, we require that the least significant $n - 1$ bits do not change between the two representations. Then we get to the next to last equation, which is only true if all of the bits in positions $n - 1$ through $m - 2$ are 1. Therefore, the sign-extension rule works. The reader may find the rule easier to grasp after studying the discussion on twos complement negation at the beginning of Section 10.3.

Fixed-Point Representation

Finally, we mention that the representations discussed in this section are sometimes referred to as fixed point. This is because the radix point (binary point) is fixed and assumed to be to the right of the rightmost digit. The programmer can use the same representation for binary fractions by scaling the numbers so that the binary point is implicitly positioned at some other location.

10.3 INTEGER ARITHMETIC

This section examines common arithmetic functions on numbers in twos complement representation.

Negation

In sign-magnitude representation, the rule for forming the negation of an integer is simple: invert the sign bit. In twos complement notation, the negation of an integer can be formed with the following rules:

1. Take the Boolean complement of each bit of the integer (including the sign bit). That is, set each 1 to 0 and each 0 to 1.
2. Treating the result as an unsigned binary integer, add 1.

This two-step process is referred to as the **twos complement operation**, or the taking of the twos complement of an integer.

$$\begin{array}{rcll} +18 & = & 00010010 & \text{(twos complement)} \\ \text{bitwise complement} & = & 11101101 & \\ & & + \quad 1 & \\ & & 11101110 = -18 & \end{array}$$

As expected, the negative of the negative of that number is itself:

$$\begin{array}{rcll} -18 & = & 11101110 & \text{(twos complement)} \\ \text{bitwise complement} & = & 00010001 & \\ & & + \quad 1 & \\ & & 00010010 = +18 & \end{array}$$

We can demonstrate the validity of the operation just described using the definition of the twos complement representation in Equation (10.2). Again, interpret an n-bit sequence of binary digits $a_{n-1}a_{n-2} \ldots a_1a_0$ as a twos complement integer A, so that its value is

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

Now form the bitwise complement, $\overline{a_{n-1}a_{n-2}} \ldots \overline{a_0}$, and, treating this as an unsigned integer, add 1. Finally, interpret the resulting n-bit sequence of binary digits as a twos complement integer B, so that its value is

$$B = -2^{n-1}\overline{a_{n-1}} + 1 + \sum_{i=0}^{n-2} 2^i \overline{a_i}$$

Now, we want $A = -B$, which means $A + B = 0$. This is easily shown to be true:

$$\begin{aligned} A + B &= -(a_{n-1} + \overline{a_{n-1}})2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i (a_i + \overline{a_i})\right) \\ &= -2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i\right) \\ &= -2^{n-1} + 1 + (2^{n-1} - 1) \\ &= -2^{n-1} + 2^{n-1} = 0 \end{aligned}$$

The preceding derivation assumes that we can first treat the bitwise complement of A as an unsigned integer for the purpose of adding 1, and then treat the result as a twos complement integer. There are two special cases to consider. First, consider $A = 0$. In that case, for an 8-bit representation:

$$\begin{array}{rcl} 0 & = & 00000000 \quad \text{(twos complement)} \\ \text{bitwise complement} & = & 11111111 \\ & & \underline{+\qquad 1} \\ & & 100000000 = 0 \end{array}$$

There is a *carry* out of the most significant bit position, which is ignored. The result is that the negation of 0 is 0, as it should be.

The second special case is more of a problem. If we take the negation of the bit pattern of 1 followed by $n - 1$ zeros, we get back the same number. For example, for 8-bit words,

$$\begin{array}{rcl} +128 & = & 10000000 \quad \text{(twos complement)} \\ \text{bitwise complement} & = & 01111111 \\ & & \underline{+\qquad 1} \\ & & 10000000 = -128 \end{array}$$

Some such anomaly is unavoidable. The number of different bit patterns in an n-bit word is $2n$, which is an even number. We wish to represent positive and negative integers and 0. If an equal number of positive and negative integers are represented (sign magnitude), then there are two representations for 0. If there is only one representation of 0 (twos complement), then there must be an unequal number of negative and positive numbers represented. In the case of twos complement, for an n-bit length, there is a representation for -2^{n-1} but not for $+2^{n-1}$.

Addition and Subtraction

Addition in twos complement is illustrated in Figure 10.3. Addition proceeds as if the two numbers were unsigned integers. The first four examples illustrate successful operations. If the result of the operation is positive, we get a positive number in twos complement form, which is the same as in unsigned-integer form. If the result of the operation is negative, we get a negative number in twos complement form. Note that, in some instances, there is a carry bit beyond the end of the word (indicated by shading), which is ignored.

On any addition, the result may be larger than can be held in the word size being used. This condition is called **overflow**. When overflow occurs, the ALU must signal this fact so that no attempt is made to use the result. To detect overflow, the following rule is observed:

OVERFLOW RULE: If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)
0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Figure 10.3 Addition of Numbers in Twos Complement Representation

Figures 10.3e and f show examples of overflow. Note that overflow can occur whether or not there is a carry.

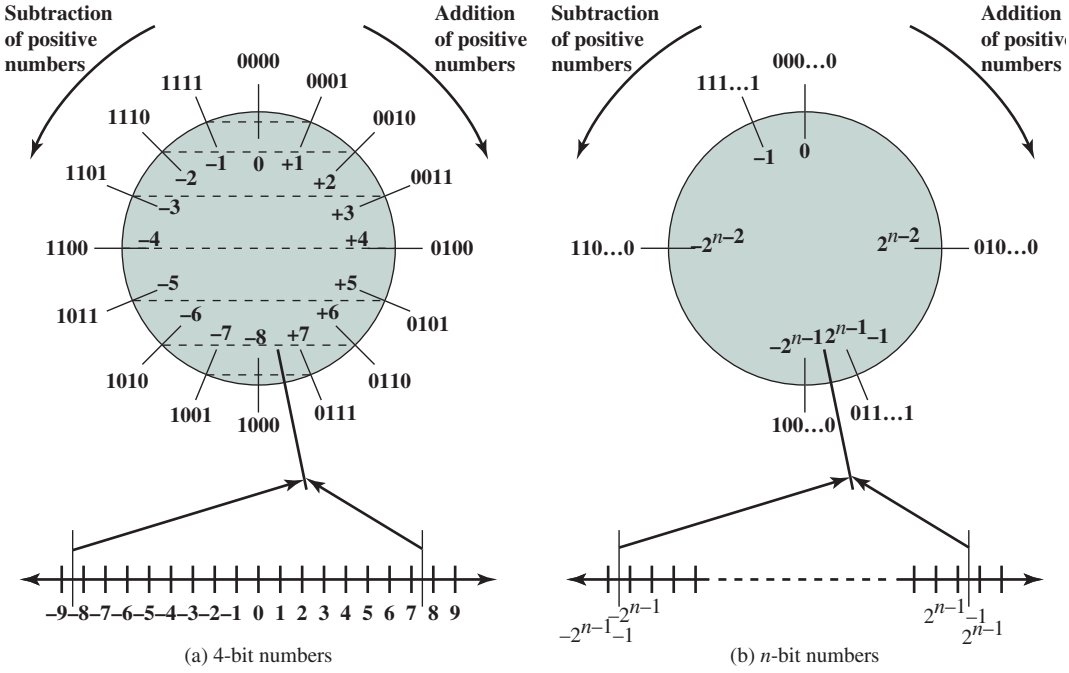
Subtraction is easily handled with the following rule:

SUBTRACTION RULE: To subtract one number (subtrahend) from another (minuend), take the twos complement (negation) of the subtrahend and add it to the minuend.

Thus, subtraction is achieved using addition, as illustrated in Figure 10.4. The last two examples demonstrate that the overflow rule still applies.

0010 = 2 +1001 = −7 1011 = −5 (a) M = 2 = 0010 S = 7 = 0111 −S = 1001	0101 = 5 +1110 = −2 10011 = 3 (b) M = 5 = 0101 S = 2 = 0010 −S = 1110
1011 = −5 +1110 = −2 11001 = −7 (c) M = −5 = 1011 S = 2 = 0010 −S = 1110	0101 = 5 +0010 = 2 0111 = 7 (d) M = 5 = 0101 S = −2 = 1110 −S = 0010
0111 = 7 +0111 = 7 1110 = Overflow (e) M = 7 = 0111 S = −7 = 1001 −S = 0111	1010 = −6 +1100 = −4 10110 = Overflow (f) M = −6 = 1010 S = 4 = 0100 −S = 1100

Figure 10.4 Subtraction of Numbers in Twos Complement Representation (M − S)



(a) 4-bit numbers (b) *n*-bit numbers

Figure 10.5 Geometric Depiction of Twos Complement Integers

Some insight into twos complement addition and subtraction can be gained by looking at a geometric depiction [BENH92], as shown in Figure 10.5. The circle in the upper half of each part of the figure is formed by selecting the appropriate segment of the number line and joining the endpoints. Note that when the numbers are laid out on a circle, the twos complement of any number is horizontally opposite that number (indicated by dashed horizontal lines). Starting at any number on the circle, we can add positive k (or subtract negative k) to that number by moving k positions clockwise, and we can subtract positive k (or add negative k) from that number by moving k positions counterclockwise. If an arithmetic operation results in traversal of the point where the endpoints are joined, an incorrect answer is given (overflow).

ALL OF the examples of Figures 10.3 and 10.4 are easily traced in the circle of Figure 10.5.

Figure 10.6 suggests the data paths and hardware elements needed to accomplish addition and subtraction. The central element is a binary adder, which is presented two numbers for addition and produces a sum and an overflow indication. The binary adder treats the two numbers as unsigned integers. (A logic implementation of an adder is given in Chapter 11.) For addition, the two numbers are presented to the adder from two registers, designated in this case as **A** and **B** registers. The result may be stored in one of these registers or in a third. The overflow indication is stored in a 1-bit overflow flag (0 = no overflow; 1 = overflow). For subtraction, the subtrahend (**B** register) is passed through a twos complementer so that its twos complement is presented to the adder. Note that Figure 10.6 only shows the

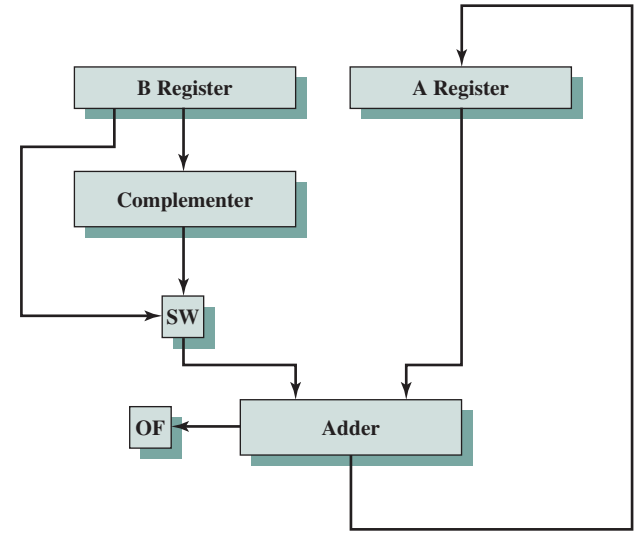


Figure 10.6 Block Diagram of Hardware for Addition and Subtraction

data paths. Control signals are needed to control whether or not the complementer is used, depending on whether the operation is addition or subtraction.

Multiplication

Compared with addition and subtraction, multiplication is a complex operation, whether performed in hardware or software. A wide variety of algorithms have been used in various computers. The purpose of this subsection is to give the reader some feel for the type of approach typically taken. We begin with the simpler problem of multiplying two unsigned (nonnegative) integers, and then we look at one of the most common techniques for multiplication of numbers in twos complement representation.

UNSIGNED INTEGERS Figure 10.7 illustrates the multiplication of unsigned binary integers, as might be carried out using paper and pencil. Several important observations can be made:

1. Multiplication involves the generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product.

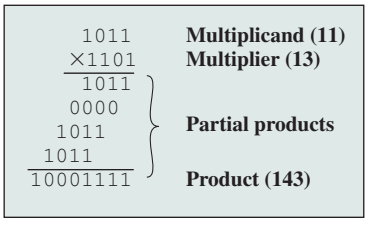
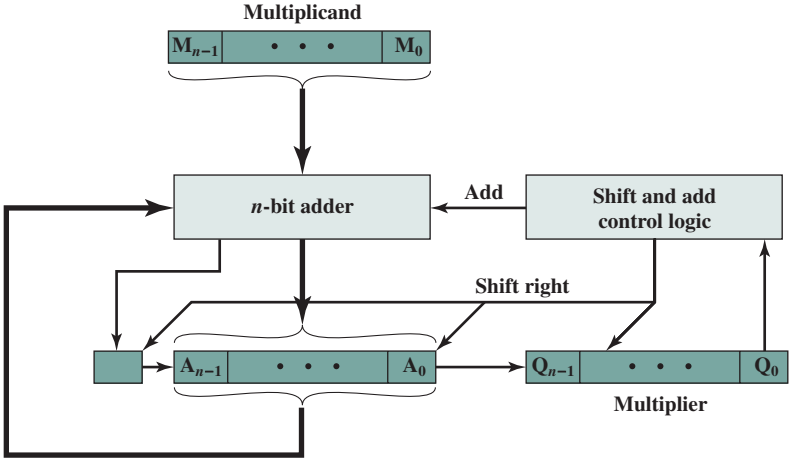


Figure 10.7 Multiplication of Unsigned Binary Integers

2. The partial products are easily defined. When the multiplier bit is 0, the partial product is 0. When the multiplier is 1, the partial product is the multiplicand.
3. The total product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product.
4. The multiplication of two n-bit binary integers results in a product of up to $2n$ bits in length (e.g., $11 \times 11 = 1001$).

Compared with the pencil-and-paper approach, there are several things we can do to make computerized multiplication more efficient. First, we can perform a running addition on the partial products rather than waiting until the end. This eliminates the need for storage of all the partial products; fewer registers are needed. Second, we can save some time on the generation of partial products. For each 1 on the multiplier, an add and a shift operation are required; but for each 0, only a shift is required.

Figure 10.8a shows a possible implementation employing these measures. The multiplier and multiplicand are loaded into two registers (Q and M). A third



(a) Block diagram

C	A	Q	M		
0	0000	1101	1011	Initial values	
0	1011	1101	1011	Add	First cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second cycle
0	1101	1111	1011	Add	Third cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth cycle
0	1000	1111	1011	Shift	

(b) Example from Figure 10.7 (product in A, Q)

Figure 10.8 Hardware Implementation of Unsigned Binary Multiplication

register, the A register, is also needed and is initially set to 0. There is also a 1-bit C register, initialized to 0, which holds a potential carry bit resulting from addition.

The operation of the multiplier is as follows. Control logic reads the bits of the multiplier one at a time. If Q_0 is 1, then the multiplicand is added to the A register and the result is stored in the A register, with the C bit used for overflow. Then all of the bits of the C, A, and Q registers are shifted to the right one bit, so that the C bit goes into A_{n-1}, A_0 goes into Q_{n-1}, and Q_0 is lost. If Q_0 is 0, then no addition is performed, just the shift. This process is repeated for each bit of the original multiplier. The resulting $2n$-bit product is contained in the A and Q registers. A flowchart of the operation is shown in Figure 10.9, and an example is given in Figure 10.8b. Note that on the second cycle, when the multiplier bit is 0, there is no add operation.

TWOS COMPLEMENT MULTIPLICATION We have seen that addition and subtraction can be performed on numbers in twos complement notation by treating them as unsigned integers. Consider

$$\begin{array}{r} 1001 \\ +\,0011 \\ \hline 1100 \end{array}$$

If these numbers are considered to be unsigned integers, then we are adding 9 (1001) plus 3 (0011) to get 12 (1100). As twos complement integers, we are adding -7(1001) to 3 (0011) to get -4(1100).

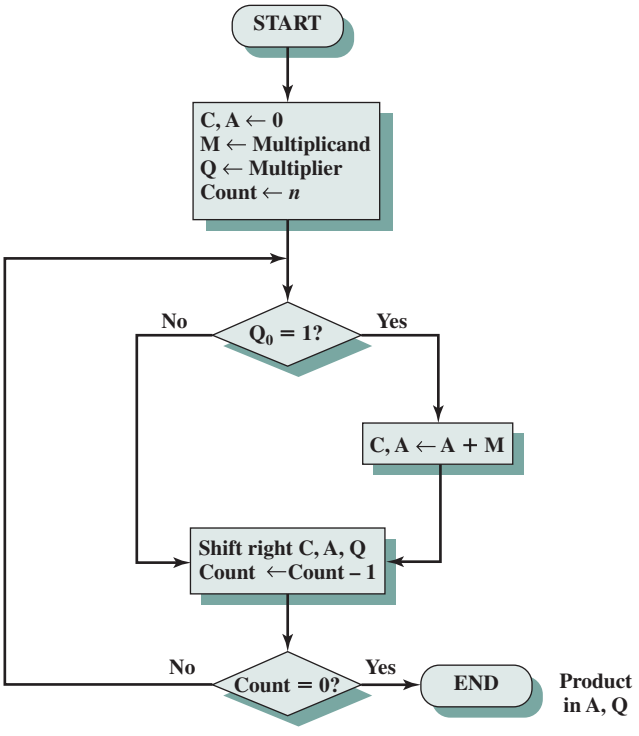


Figure 10.9 Flowchart for Unsigned Binary Multiplication

```
      1011
    × 1101
  --------
  00001011   1011 × 1 × 2^0
  00000000   1011 × 0 × 2^1
  00101100   1011 × 1 × 2^2
  01011000   1011 × 1 × 2^3
  --------
  10001111
```

Figure 10.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

Unfortunately, this simple scheme will not work for multiplication. To see this, consider again Figure 10.7. We multiplied 11 (1011) by 13 (1101) to get 143 (10001111). If we interpret these as twos complement numbers, we have −5(1011) times −3 (1101) equals −113 (10001111). This example demonstrates that straightforward multiplication will not work if both the multiplicand and multiplier are negative. In fact, it will not work if either the multiplicand or the multiplier is negative. To justify this statement, we need to go back to Figure 10.7 and explain what is being done in terms of operations with powers of 2. Recall that any unsigned binary number can be expressed as a sum of powers of 2. Thus,

$$1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 2^3 + 2^2 + 2^0$$

Further, the multiplication of a binary number by 2^n is accomplished by shifting that number to the left n bits. With this in mind, Figure 10.10 recasts Figure 10.7 to make the generation of partial products by multiplication explicit. The only difference in Figure 10.10 is that it recognizes that the partial products should be viewed as $2n$-bit numbers generated from the n-bit multiplicand.

Thus, as an unsigned integer, the 4-bit multiplicand 1011 is stored in an 8-bit word as 00001011. Each partial product (other than that for 2^0) consists of this number shifted to the left, with the unoccupied positions on the right filled with zeros (e.g., a shift to the left of two places yields 00101100).

Now we can demonstrate that straightforward multiplication will not work if the multiplicand is negative. The problem is that each contribution of the negative multiplicand as a partial product must be a negative number on a $2n$-bit field; the sign bits of the partial products must line up. This is demonstrated in Figure 10.11, which shows that multiplication of 1001 by 0011. If these are treated as unsigned integers, the multiplication of $9 \times 3 = 27$ proceeds simply. However, if 1001 is interpreted

```
    1001  (9)                  1001  (-7)
  × 0011  (3)                × 0011  (3)
  --------                   --------
  00001001  1001 × 2^0       11111001  (-7) × 2^0 = (-7)
  00010010  1001 × 2^1       11110010  (-7) × 2^1 = (-14)
  --------                   --------
  00011011  (27)             11101011  (-21)
```

(a) Unsigned integers (b) Twos complement integers

Figure 10.11 Comparison of Multiplication of Unsigned and Twos Complement Integers

as the twos complement value −7, then each partial product must be a negative twos complement number of 2*n* (8) bits, as shown in Figure 10.11b. Note that this is accomplished by padding out each partial product to the left with binary 1s.

If the multiplier is negative, straightforward multiplication also will not work. The reason is that the bits of the multiplier no longer correspond to the shifts or multiplications that must take place. For example, the 4-bit decimal number −3 is written 1101 in twos complement. If we simply took partial products based on each bit position, we would have the following correspondence:

$$1101 \leftrightarrow -(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) = -(2^3 + 2^2 + 2^0)$$

In fact, what is desired is $-(2^1 + 2^0)$. So this multiplier cannot be used directly in the manner we have been describing.

There are a number of ways out of this dilemma. One would be to convert both multiplier and multiplicand to positive numbers, perform the multiplication, and then take the twos complement of the result if and only if the sign of the two original numbers differed. Implementers have preferred to use techniques that do not require this final transformation step. One of the most common of these is Booth's algorithm [BOOT51]. This algorithm also has the benefit of speeding up the multiplication process, relative to a more straightforward approach.

Booth's algorithm is depicted in Figure 10.12 and can be described as follows. As before, the multiplier and multiplicand are placed in the Q and M registers,

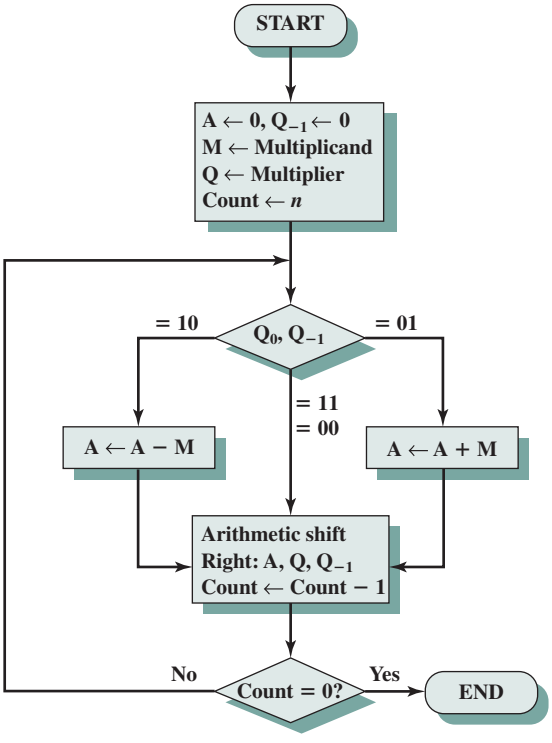


Figure 10.12 Booth's Algorithm for Twos Complement Multiplication

A	Q	Q_{-1}	M		
0000	0011	0	0111	**Initial values**	
1001	0011	0	0111	**A←A – M**	**First cycle**
1100	1001	1	0111	**Shift**	
1110	0100	1	0111	**Shift**	**Second cycle**
0101	0100	1	0111	**A←A + M**	**Third cycle**
0010	1010	0	0111	**Shift**	
0001	0101	0	0111	**Shift**	**Fourth cycle**

Figure 10.13 Example of Booth's Algorithm (7 × 3)

respectively. There is also a 1-bit register placed logically to the right of the least significant bit (Q_0) of the Q register and designated Q_{-1}; its use is explained shortly. The results of the multiplication will appear in the A and Q registers. A and Q_{-1} are initialized to 0. As before, control logic scans the bits of the multiplier one at a time. Now, as each bit is examined, the bit to its right is also examined. If the two bits are the same (1–1 or 0–0), then all of the bits of the A, Q, and Q_{-1} registers are shifted to the right 1 bit. If the two bits differ, then the multiplicand is added to or subtracted from the A register, depending on whether the two bits are 0–1 or 1–0. Following the addition or subtraction, the right shift occurs. In either case, the right shift is such that the leftmost bit of A, namely A_{n-1}, not only is shifted into A_{n-2}, but also remains in A_{n-1}. This is required to preserve the sign of the number in A and Q. It is known as an **arithmetic shift**, because it preserves the sign bit.

Figure 10.13 shows the sequence of events in Booth's algorithm for the multiplication of 7 by 3. More compactly, the same operation is depicted in Figure 10.14a. The rest of Figure 10.14 gives other examples of the algorithm. As can be seen, it works with any combination of positive and negative numbers. Note also the efficiency of the algorithm. Blocks of 1s or 0s are skipped over, with an average of only one addition or subtraction per block.

```
   0111
 × 0011    (0)
11111001   1-0
0000000    1-1
000111     0-1
00010101   (21)
```

(a) (7) × (3) = (21)

```
   0111
 × 1101    (0)
11111001   1-0
0000111    0-1
111001     1-0
11101011   (-21)
```

(b) (7) × (−3) = (−21)

```
   1001
 × 0011    (0)
00000111   1-0
0000000    1-1
111001     0-1
11101011   (-21)
```

(c) (−7) × (3) = (−21)

```
   1001
 × 1101    (0)
00000111   1-0
1111001    0-1
000111     1-0
00010101   (21)
```

(d) (−7) × (−3) = (21)

Figure 10.14 Examples Using Booth's Algorithm

Why does Booth's algorithm work? Consider first the case of a positive multiplier. In particular, consider a positive multiplier consisting of one block of 1s surrounded by 0s (e.g., 00011110). As we know, multiplication can be achieved by adding appropriately shifted copies of the multiplicand:

$$\begin{aligned} \text{M} \times (00011110) &= \text{M} \times (2^4 + 2^3 + 2^2 + 2^1) \\ &= \text{M} \times (16 + 8 + 4 + 2) \\ &= \text{M} \times 30 \end{aligned}$$

The number of such operations can be reduced to two if we observe that

$$2^n + 2^{n-1} + \cdots + 2^{n-K} = 2^{n+1} - 2^{n-K} \tag{10.3}$$

$$\begin{aligned} \text{M} \times (00011110) &= \text{M} \times (2^5 - 2^1) \\ &= \text{M} \times (32 - 2) \\ &= \text{M} \times 30 \end{aligned}$$

So the product can be generated by one addition and one subtraction of the multiplicand. This scheme extends to any number of blocks of 1s in a multiplier, including the case in which a single 1 is treated as a block.

$$\begin{aligned} \text{M} \times (01111010) &= \text{M} \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1) \\ &= \text{M} \times (2^7 - 2^3 + 2^2 - 2^1) \end{aligned}$$

Booth's algorithm conforms to this scheme by performing a subtraction when the first 1 of the block is encountered (1–0) and an addition when the end of the block is encountered (0–1).

To show that the same scheme works for a negative multiplier, we need to observe the following. Let X be a negative number in twos complement notation:

$$\text{Representation of } X = \{1x_{n-2}x_{n-3} \ldots x_1x_0\}$$

Then the value of X can be expressed as follows:

$$X = -2^{n-1} + (x_{n-2} \times 2^{n-2}) + (x_{n-3} \times 2^{n-3}) + \cdots (x_1 \times 2^1) + (x_0 \times 2^0) \tag{10.4}$$

The reader can verify this by applying the algorithm to the numbers in Table 10.2.

The leftmost bit of X is 1, because X is negative. Assume that the leftmost 0 is in the kth position. Thus, X is of the form

$$\text{Representation of } X = \{111 \ldots 10x_{k-1}x_{k-2} \ldots x_1x_0\} \tag{10.5}$$

Then the value of X is

$$X = -2^{n-1} + 2^{n-2} + \cdots + 2^{k+1} + (x_{k-1} \times 2^{k-1}) + \cdots + (x_0 \times 2^0) \tag{10.6}$$

From Equation (10.3), we can say that

$$2^{n-2} + 2^{n-3} + \cdots + 2^{k-1} = 2^{n-1} - 2^{k-1}$$

Rearranging

$$-2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2^{k+1} = -2^{k+1} \quad \textbf{(10.7)}$$

Substituting Equation (10.7) into Equation (10.6), we have

$$X = -2^{k+1} + (x_{k-1} \times 2^{k-1}) + \cdots + (x_0 \times 2^0) \quad \textbf{(10.8)}$$

At last we can return to Booth's algorithm. Remembering the representation of X [Equation (10.5)], it is clear that all of the bits from x_0 up to the leftmost 0 are handled properly because they produce all of the terms in Equation (10.8) but (-2^{k+1}) and thus are in the proper form. As the algorithm scans over the leftmost 0 and encounters the next 1 (2^{k+1}), a 1–0 transition occurs and a subtraction takes place (-2^{k+1}). This is the remaining term in Equation (10.8).

As an example, consider the multiplication of some multiplicand by (-6). In twos complement representation, using an 8-bit word, (-6) is represented as 11111010. By Equation (10.4), we know that

$$-6 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1$$

which the reader can easily verify. Thus,

$$\text{M} \times (11111010) = \text{M} \times (-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1)$$

Using Equation (10.7),

$$\text{M} \times (11111010) = \text{M} \times (-2^3 + 2^1)$$

which the reader can verify is still $\text{M} \times (-6)$. Finally, following our earlier line of reasoning,

$$\text{M} \times (11111010) = \text{M} \times (-2^3 + 2^2 - 2^1)$$

We can see that Booth's algorithm conforms to this scheme. It performs a subtraction when the first 1 is encountered (10), an addition when (01) is encountered, and finally another subtraction when the first 1 of the next block of 1s is encountered. Thus, Booth's algorithm performs fewer additions and subtractions than a more straightforward algorithm.

Division

Division is somewhat more complex than multiplication but is based on the same general principles. As before, the basis for the algorithm is the paper-and-pencil approach, and the operation involves repetitive shifting and addition or subtraction.

Figure 10.15 shows an example of the long division of unsigned binary integers. It is instructive to describe the process in detail. First, the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor; this is referred to as the divisor being able to divide the number. Until this event occurs, 0s are placed in the quotient from left to right. When the event occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a *partial remainder.*

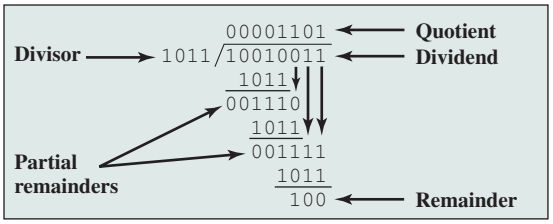


Figure 10.15 Example of Division of Unsigned Binary Integers

From this point on, the division follows a cyclic pattern. At each cycle, additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the divisor. As before, the divisor is subtracted from this number to produce a new partial remainder. The process continues until all the bits of the dividend are exhausted.

Figure 10.16 shows a machine algorithm that corresponds to the long division process. The divisor is placed in the M register, the dividend in the Q register. At

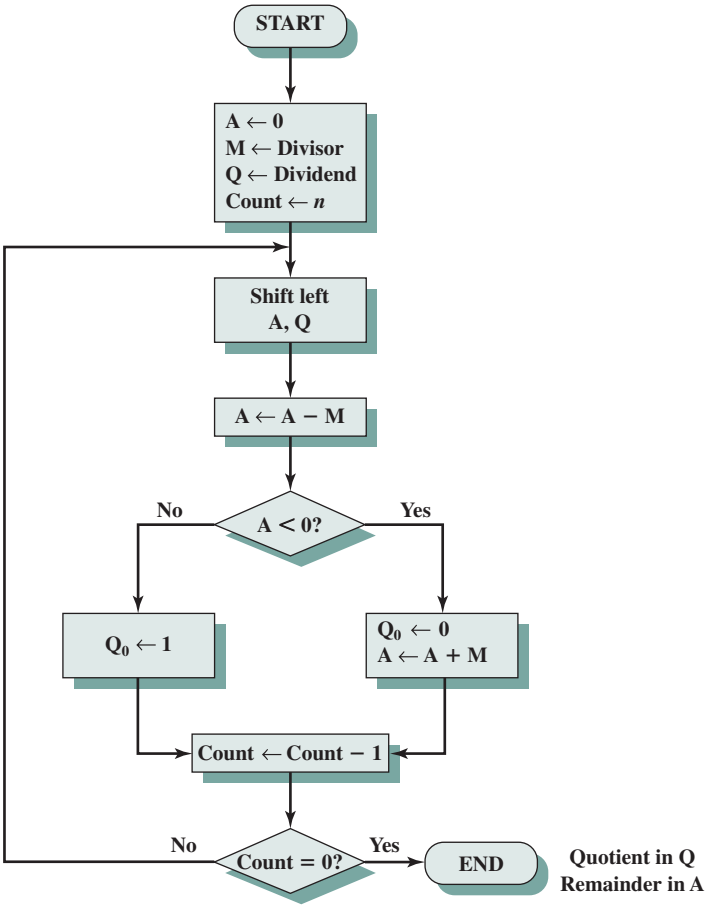


Figure 10.16 Flowchart for Unsigned Binary Division

A	Q	
0000	0111	**Initial value**
0000	1110	**Shift**
1101		**Use twos complement of 0011 for subtraction**
1101		**Subtract**
0000	1110	**Restore, set $Q_0 = 0$**
0001	1100	**Shift**
1101		
1110		**Subtract**
0001	1100	**Restore, set $Q_0 = 0$**
0011	1000	**Shift**
1101		
0000	1001	**Subtract, set $Q_0 = 1$**
0001	0010	**Shift**
1101		
1110		**Subtract**
0001	0010	**Restore, set $Q_0 = 0$**

Figure 10.17 Example of Restoring Twos Complement Division (7/3)

each step, the A and Q registers together are shifted to the left 1 bit. M is subtracted from A to determine whether A divides the partial remainder.[3] If it does, then Q_0 gets a 1 bit. Otherwise, Q_0 gets a 0 bit and M must be added back to A to restore the previous value. The count is then decremented, and the process continues for n steps. At the end, the quotient is in the Q register and the remainder is in the A register.

This process can, with some difficulty, be extended to negative numbers. We give here one approach for twos complement numbers. An example of this approach is shown in Figure 10.17.

The algorithm assumes that the divisor V and the dividend D are positive and that $|V| < |D|$. If $|V| = |D|$, then the quotient $Q = 1$ and the remainder $R = 0$. If $|V| > |D|$, then $Q = 0$ and $R = D$. The algorithm can be summarized as follows:

1. Load the twos complement of the divisor into the M register; that is, the M register contains the negative of the divisor. Load the dividend into the A, Q registers. The dividend must be expressed as a $2n$-bit positive number. Thus, for example, the 4-bit 0111 becomes 00000111.
2. Shift A, Q left 1 bit position.
3. Perform $A \leftarrow A - M$. This operation subtracts the divisor from the contents of A.
4. **a.** If the result is nonnegative (most significant bit of $A = 0$), then set $Q_0 \leftarrow 1$.
   b. If the result is negative (most significant bit of $A = 1$), then set $Q_0 \leftarrow 0$. and restore the previous value of A.
5. Repeat steps 2 through 4 as many times as there are bit positions in Q.
6. The remainder is in A and the quotient is in Q.

[3]This is subtraction of unsigned integers. A result that requires a borrow out of the most significant bit is a negative result.

To deal with negative numbers, we recognize that the remainder is defined by

$$D = Q \times V + R$$

That is, the remainder is the value of R needed for the preceding equation to be valid. Consider the following examples of integer division with all possible combinations of signs of D and V:

$$\begin{array}{llllll} D = 7 & V = 3 & \Rightarrow & Q = 2 & R = 1 \\ D = 7 & V = -3 & \Rightarrow & Q = -2 & R = 1 \\ D = -7 & V = 3 & \Rightarrow & Q = -2 & R = -1 \\ D = -7 & V = -3 & \Rightarrow & Q = 2 & R = -1 \end{array}$$

The reader will note from Figure 10.17 that $(-7)/(3)$ and $(7)/(-3)$ produce different remainders. We see that the magnitudes of Q and R are unaffected by the input signs and that the signs of Q and R are easily derivable from the signs of D and V. Specifically, $\text{sign}(R) = \text{sign}(D)$ and $\text{sign}(Q) = \text{sign}(D) \times \text{sign}(V)$. Hence, one way to do twos complement division is to convert the operands into unsigned values and, at the end, to account for the signs by complementation where needed. This is the method of choice for the restoring division algorithm [PARH10].

10.4 FLOATING-POINT REPRESENTATION

Principles

With a fixed-point notation (e.g., twos complement) it is possible to represent a range of positive and negative integers centered on or near 0. By assuming a fixed binary or radix point, this format allows the representation of numbers with a fractional component as well.

This approach has limitations. Very large numbers cannot be represented, nor can very small fractions. Furthermore, the fractional part of the quotient in a division of two large numbers could be lost.

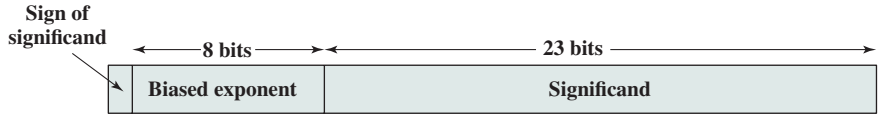
For decimal numbers, we get around this limitation by using scientific notation. Thus, 976,000,000,000,000 can be represented as 9.76×10^{14}, and 0.0000000000000976 can be represented as 9.76×10^{-14}, What we have done, in effect, is dynamically to slide the decimal point to a convenient location and use the exponent of 10 to keep track of that decimal point. This allows a range of very large and very small numbers to be represented with only a few digits.

This same approach can be taken with binary numbers. We can represent a number in the form

$$\pm \mathrm{S} \times \mathrm{B}^{\pm \mathrm{E}}$$

This number can be stored in a binary word with three fields:

- Sign: plus or minus
- Significand S
- Exponent E



(a) Format

$$
\begin{aligned}
1.1010001 \times 2^{10100} &= 0\ 10010011\ 10100010000000000000000 = 1.6328125 \times 2^{20} \\
-1.1010001 \times 2^{10100} &= 1\ 10010011\ 10100010000000000000000 = -1.6328125 \times 2^{20} \\
1.1010001 \times 2^{-10100} &= 0\ 01101011\ 10100010000000000000000 = 1.6328125 \times 2^{-20} \\
-1.1010001 \times 2^{-10100} &= 1\ 01101011\ 10100010000000000000000 = -1.6328125 \times 2^{-20}
\end{aligned}
$$

(b) Examples

Figure 10.18 Typical 32-Bit Floating-Point Format

The **base** B is implicit and need not be stored because it is the same for all numbers. Typically, it is assumed that the radix point is to the right of the leftmost, or most significant, bit of the significand. That is, there is one bit to the left of the radix point.

The principles used in representing binary floating-point numbers are best explained with an example. Figure 10.18a shows a typical 32-bit floating-point format. The leftmost bit stores the **sign** of the number (0 = positive, 1 = negative). The **exponent** value is stored in the next 8 bits. The representation used is known as a **biased representation**. A fixed value, called the bias, is subtracted from the field to get the true exponent value. Typically, the bias equals $(2^{k-1} - 1)$, where k is the number of bits in the binary exponent. In this case, the 8-bit field yields the numbers 0 through 255. With a bias of 127 $(2^7 - 1)$, the true exponent values are in the range -127 to $+128$. In this example, the base is assumed to be 2.

Table 10.2 shows the biased representation for 4-bit integers. Note that when the bits of a biased representation are treated as unsigned integers, the relative magnitudes of the numbers do not change. For example, in both biased and unsigned representations, the largest number is 1111 and the smallest number is 0000. This is not true of sign-magnitude or twos complement representation. An advantage of biased representation is that nonnegative floating-point numbers can be treated as integers for comparison purposes.

The final portion of the word (23 bits in this case) is the **significand**.[4]

Any floating-point number can be expressed in many ways.

The following are equivalent, where the significand is expressed in binary form:

$$0.110 \times 2^5$$
$$110 \times 2^2$$
$$0.0110 \times 2^6$$

To simplify operations on floating-point numbers, it is typically required that they be normalized. A **normal number** is one in which the most significant digit of the

[4]The term **mantissa**, sometimes used instead of *significand,* is considered obsolete. *Mantissa* also means "the fractional part of a logarithm," so is best avoided in this context.

significand is nonzero. For base 2 representation, a normal number is therefore one in which the most significant bit of the significand is one. As was mentioned, the typical convention is that there is one bit to the left of the radix point. Thus, a normal nonzero number is one in the form

$$\pm 1.\text{bbb} \ldots \text{ b} \times 2^{\pm \text{E}}$$

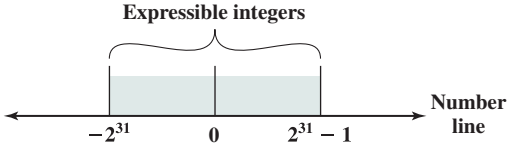
where b is either binary digit (0 or 1). Because the most significant bit is always one, it is unnecessary to store this bit; rather, it is implicit. Thus, the 23-bit field is used to store a 24-bit significand with a value in the half open interval [1, 2). Given a number that is not normal, the number may be normalized by shifting the radix point to the right of the leftmost 1 bit and adjusting the exponent accordingly.

Figure 10.18b gives some examples of numbers stored in this format. For each example, on the left is the binary number; in the center is the corresponding bit pattern; on the right is the decimal value. Note the following features:

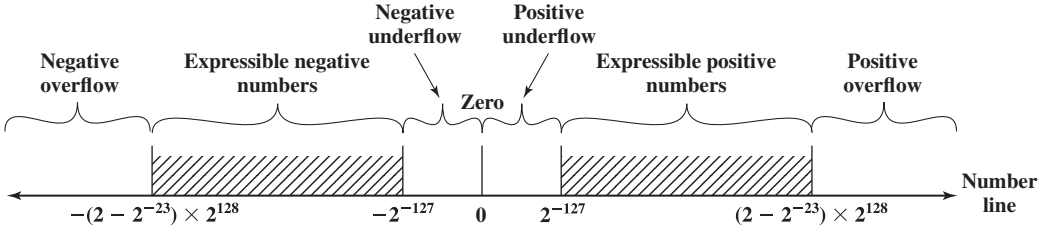
- The sign is stored in the first bit of the word.
- The first bit of the true significand is always 1 and need not be stored in the significand field.
- The value 127 is added to the true exponent to be stored in the exponent field.
- The base is 2.

For comparison, Figure 10.19 indicates the range of numbers that can be represented in a 32-bit word. Using twos complement integer representation, all of the integers from -2^{31} to $2^{31} - 1$ can be represented, for a total of 2^{32} different numbers. With the example floating-point format of Figure 10.18, the following ranges of numbers are possible:

- Negative numbers between $-(2 - 2^{-23}) \times 2^{128}$ and -2^{-127}
- Positive numbers between 2^{-127} and $(2 - 2^{-23}) \times 2^{128}$



(a) Twos complement integers



(b) Floating-point numbers

Figure 10.19 Expressible Numbers in Typical 32-Bit Formats

Five regions on the number line are not included in these ranges:

- Negative numbers less than $-(2 - 2^{-23}) \times 2^{128}$, called **negative overflow**
- Negative numbers greater than 2^{-127}, called **negative underflow**
- Zero
- Positive numbers less than 2^{-127}, called **positive underflow**
- Positive numbers greater than $(2 - 2^{-23}) \times 2^{128}$, called **positive overflow**

The representation as presented will not accommodate a value of 0. However, as we shall see, actual floating-point representations include a special bit pattern to designate zero. Overflow occurs when an arithmetic operation results in an absolute value greater than can be expressed with an exponent of 128 (e.g., $2^{120} \times 2^{100} = 2^{220}$). Underflow occurs when the fractional magnitude is too small (e.g., $2^{-120} \times 2^{-100} = 2^{-220}$). Underflow is a less serious problem because the result can generally be satisfactorily approximated by 0.

It is important to note that we are not representing more individual values with floating-point notation. The maximum number of different values that can be represented with 32 bits is still 2^{32}. What we have done is to spread those numbers out in two ranges, one positive and one negative. In practice, most floating-point numbers that one would wish to represent are represented only approximately. However, for moderate sized integers, the representation is exact.

Also, note that the numbers represented in floating-point notation are not spaced evenly along the number line, as are fixed-point numbers. The possible values get closer together near the origin and farther apart as you move away, as shown in Figure 10.20. This is one of the trade-offs of floating-point math: Many calculations produce results that are not exact and have to be rounded to the nearest value that the notation can represent.

In the type of format depicted in Figure 10.18, there is a trade-off between range and precision. The example shows 8 bits devoted to the exponent and 23 to the significand. If we increase the number of bits in the exponent, we expand the range of expressible numbers. But because only a fixed number of different values can be expressed, we have reduced the density of those numbers and therefore the precision. The only way to increase both range and precision is to use more bits. Thus, most computers offer, at least, single-precision numbers and doubleprecision numbers. For example, a processor could support a single-precision format of 64 bits, and a double-precision format of 128 bits.

So there is a trade-off between the number of bits in the exponent and the number of bits in the significand. But it is even more complicated than that. The implied base of the exponent need not be 2. The IBM S/390 architecture, for example, uses a base of 16 [ANDE67b]. The format consists of a 7-bit exponent and a 24-bit significand.

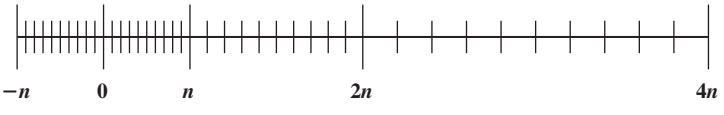


Figure 10.20 Density of Floating-Point Numbers

In the IBM base-16 format,

$$0.11010001 \times 2^{10100} = 0.11010001 \times 16^{101}$$

and the exponent is stored to represent 5 rather than 20.

The advantage of using a larger exponent is that a greater range can be achieved for the same number of exponent bits. But remember, we have not increased the number of different values that can be represented. Thus, for a fixed format, a larger exponent base gives a greater range at the expense of less precision.

IEEE Standard for Binary Floating-Point Representation

The most important floating-point representation is defined in IEEE Standard 754, adopted in 1985 and revised in 2008. This standard was developed to facilitate the portability of programs from one processor to another and to encourage the development of sophisticated, numerically oriented programs. The standard has been widely adopted and is used on virtually all contemporary processors and arithmetic coprocessors. IEEE 754-2008 covers both binary and decimal floating-point representations. In this chapter, we deal only with binary representations.

IEEE 754-2008 defines the following different types of floating-point formats:

- **Arithmetic format:** All the mandatory operations defined by the standard are supported by the format. The format may be used to represent floating-point operands or results for the operations described in the standard.
- **Basic format:** This format covers five floating-point representations, three binary and two decimal, whose encodings are specified by the standard, and which can be used for arithmetic. At least one of the basic formats is implemented in any conforming implementation.
- **Interchange format:** A fully specified, fixed-length binary encoding that allows data interchange between different platforms and that can be used for storage.

The three basic binary formats have bit lengths of 32, 64, and 128 bits, with exponents of 8, 11, and 15 bits, respectively (Figure 10.21). Table 10.3 summarizes the characteristics of the three formats. The two basic decimal formats have bit lengths of 64 and 128 bits. All of the basic formats are also arithmetic format types (can be used for arithmetic operations) and interchange format types (platform independent).

Several other formats are specified in the standard. The binary16 format is only an interchange format and is intended for storage of values when higher precision is not required. The binary{k} format and the decimal{k} format are interchange formats with total length k bits and with defined lengths for the significand and exponent. The format must be a multiple of 32 bits; thus formats are defined for $k = 160, 192$, and so on. These two families of formats are also arithmetic formats.

In addition, the standard defines **extended precision formats**, which extend a supported basic format by providing additional bits in the exponent (extended range) and in the significand (extended precision). The exact format

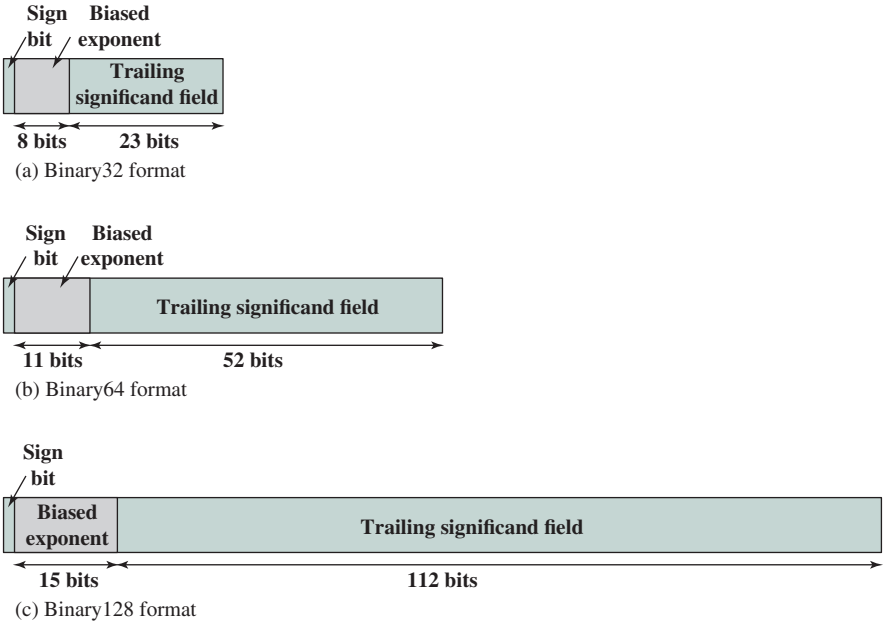


Figure 10.21 IEEE 754 Formats

is implementation dependent, but the standard places certain constraints on the length of the exponent and significand. These formats are arithmetic format types but not interchange format types. The extended formats are to be used for intermediate calculations. With their greater precision, the extended formats lessen the

Table 10.3 IEEE 754 Format Parameters

Parameter	Format		
	Binary32	Binary64	Binary128
Storage width (bits)	32	64	128
Exponent width (bits)	8	11	15
Exponent bias	127	1023	16383
Maximum exponent	127	1023	16383
Minimum exponent	−126	−1022	−16382
Approx normal number range (base 10)	$10^{-38}, 10^{+38}$	$10^{-308}, 10^{+308}$	$10^{-4932}, 10^{+4932}$
Trailing significand width (bits)*	23	52	112
Number of exponents	254	2046	32766
Number of fractions	2^{23}	2^{52}	2^{112}
Number of values	1.98×2^{31}	1.99×2^{63}	1.99×2^{128}
Smallest positive normal number	2^{-126}	2^{-1022}	2^{-16362}
Largest positive normal number	$2^{128} - 2^{104}$	$2^{1024} - 2^{971}$	$2^{16384} - 2^{16271}$
Smallest subnormal magnitude	2^{-149}	2^{-1074}	2^{-16494}

Note: * Not including implied bit and not including sign bit.

chance of a final result that has been contaminated by excessive roundoff error; with their greater range, they also lessen the chance of an intermediate overflow aborting a computation whose final result would have been representable in a basic format. An additional motivation for the extended format is that it affords some of the benefits of a larger basic format without incurring the time penalty usually associated with higher precision.

Finally, IEEE 754-2008 defines an **extendable precision format** as a format with a precision and range that are defined under user control. Again, these formats may be used for intermediate calculations, but the standard places no constraint or format or length.

Table 10.4 shows the relationship between defined formats and format types.

Not all bit patterns in the IEEE formats are interpreted in the usual way; instead, some bit patterns are used to represent special values. Table 10.5 indicates the values assigned to various bit patterns. The exponent values of all zeros (0 bits) and all ones (1 bits) define special values. The following classes of numbers are represented:

- For exponent values in the range of 1 through 254 for 32-bit format, 1 through 2046 for 64-bit format, and 1 through 16382, normal nonzero floating-point numbers are represented. The exponent is biased, so that the range of exponents is −126 through +127 for 32-bit format, and so on. A normal number requires a 1 bit to the left of the binary point; this bit is implied, giving an effective 24-bit, 53-bit, or 113-bit significand. Because one of the bits is implied, the corresponding field in the binary format is referred to as the **trailing significand field**.
- An exponent of zero together with a fraction of zero represents positive or negative zero, depending on the sign bit. As was mentioned, it is useful to have an exact value of 0 represented.

Table 10.4 IEEE Formats

Format	Format Type		
	Arithmetic Format	Basic Format	Interchange Format
binary16			X
binary32	X	X	X
binary64	X	X	X
binary128	X	X	X
binary{*k*} **(*k* = *n* × 32 for *n* > 4)**	X		X
decimal64	X	X	X
decimal128	X	X	X
decimal{*k*} **(*k* = *n* × 32 for *n* > 4)**	X		X
extended precision	X		
extendable precision	X		

Table 10.5 Interpretation of IEEE 754 Floating-Point Numbers

(a) binary32 format

	Sign	Biased Exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	−0
plus infinity	0	all 1s	0	∞
minus infinity	1	all 1s	0	−∞
quiet NaN	0 or 1	all 1s	≠ 0; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	≠ 0; first bit = 0	sNaN
positive normal nonzero	0	0 < e < 225	f	$2^{e-127}(1.f)$
negative normal nonzero	1	0 < e < 225	f	$-2^{e-127}(1.f)$
positive subnormal	0	0	f ≠ 0	$2^{e-126}(0.f)$
negative subnormal	1	0	f ≠ 0	$-2^{e-126}(0.f)$

(b) binary64 format

	Sign	Biased Exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	−0
plus infinity	0	all 1s	0	∞
minus infinity	1	all 1s	0	−∞
quiet NaN	0 or 1	all 1s	≠ 0; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	≠ 0; first bit = 0	sNaN
positive normal nonzero	0	0 < e < 2047	f	$2^{e-1023}(1.f)$
negative normal nonzero	1	0 < e < 2047	f	$-2^{e-1023}(1.f)$
positive subnormal	0	0	f ≠ 0	$2^{e-1022}(0.f)$
negative subnormal	1	0	f ≠ 0	$-2^{e-1022}(0.f)$

(c) binary128 format

	Sign	Biased Exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	−0
plus infinity	0	all 1s	0	∞
minus infinity	1	all 1s	0	−∞
quiet NaN	0 or 1	all 1s	≠ 0; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	≠ 0; first bit = 0	sNaN
positive normal nonzero	0	all 1s	f	$2^{e-16383}(1.f)$
negative normal nonzero	1	all 1s	f	$-2^{e-16383}(1.f)$
positive subnormal	0	0	f ≠ 0	$2^{e-16383}(0.f)$
negative subnormal	1	0	f ≠ 0	$-2^{e-16383}(0.f)$

- An exponent of all ones together with a fraction of zero represents positive or negative infinity, depending on the sign bit. It is also useful to have a representation of infinity. This leaves it up to the user to decide whether to treat overflow as an error condition or to carry the value ∞ and proceed with whatever program is being executed.
- An exponent of zero together with a nonzero fraction represents a subnormal number. In this case, the bit to the left of the binary point is zero and the true exponent is -126 or -1022. The number is positive or negative depending on the sign bit.
- An exponent of all ones together with a nonzero fraction is given the value NaN, which means *Not a Number*, and is used to signal various exception conditions.

The significance of subnormal numbers and NaNs is discussed in Section 10.5.

10.5 FLOATING-POINT ARITHMETIC

Table 10.6 summarizes the basic operations for floating-point arithmetic. For addition and subtraction, it is necessary to ensure that both operands have the same exponent value. This may require shifting the radix point on one of the operands to achieve alignment. Multiplication and division are more straightforward.

A floating-point operation may produce one of these conditions:

- **Exponent overflow:** A positive exponent exceeds the maximum possible exponent value. In some systems, this may be designated as $+\infty$ or $-\infty$.
- **Exponent underflow:** A negative exponent is less than the minimum possible exponent value (e.g., -200 is less than -127). This means that the number is too small to be represented, and it may be reported as 0.

Table 10.6 Floating-Point Numbers and Arithmetic Operations

Floating-Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$\left.\begin{array}{l} X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E} \\ X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E} \end{array}\right\} X_E \leq Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \times B^{X_E - Y_E}$

Examples:

$X = 0.3 \times 10^2 = 30$

$Y = 0.2 \times 10^3 = 200$

$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$

$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$

$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$

$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$

- **Significand underflow:** In the process of aligning significands, digits may flow off the right end of the significand. As we will discuss, some form of rounding is required.
- **Significand overflow:** The addition of two significands of the same sign may result in a carry out of the most significant bit. This can be fixed by realignment, as we will explain.

Addition and Subtraction

In floating-point arithmetic, addition and subtraction are more complex than multiplication and division. This is because of the need for alignment. There are four basic phases of the algorithm for addition and subtraction:

1. Check for zeros.
2. Align the significands.
3. Add or subtract the significands.
4. Normalize the result.

A typical flowchart is shown in Figure 10.22. A step-by-step narrative highlights the main functions required for floating-point addition and subtraction. We assume a format similar to those of Figure 10.21. For the addition or subtraction operation, the two operands must be transferred to registers that will be used by the ALU. If the floating-point format includes an implicit significand bit, that bit must be made explicit for the operation.

Phase 1. Zero check: Because addition and subtraction are identical except for a sign change, the process begins by changing the sign of the subtrahend if it is a subtract operation. Next, if either operand is 0, the other is reported as the result.

Phase 2. Significand alignment: The next phase is to manipulate the numbers so that the two exponents are equal.

To see the need for aligning exponents, consider the following decimal addition:

$$(123 \times 10^0) + (456 \times 10^{-2})$$

Clearly, we cannot just add the significands. The digits must first be set into equivalent positions, that is, the 4 of the second number must be aligned with the 3 of the first. Under these conditions, the two exponents will be equal, which is the mathematical condition under which two numbers in this form can be added. Thus,

$$(123 \times 10^0) + (456 \times 10^{-2}) = (123 \times 10^0) + (4.56 \times 10^0) = 127.56 \times 10^0$$

Alignment may be achieved by shifting either the smaller number to the right (increasing its exponent) or shifting the larger number to the left. Because either operation may result in the loss of digits, it is the smaller number that is shifted; any digits that are lost are therefore of relatively small significance. The alignment

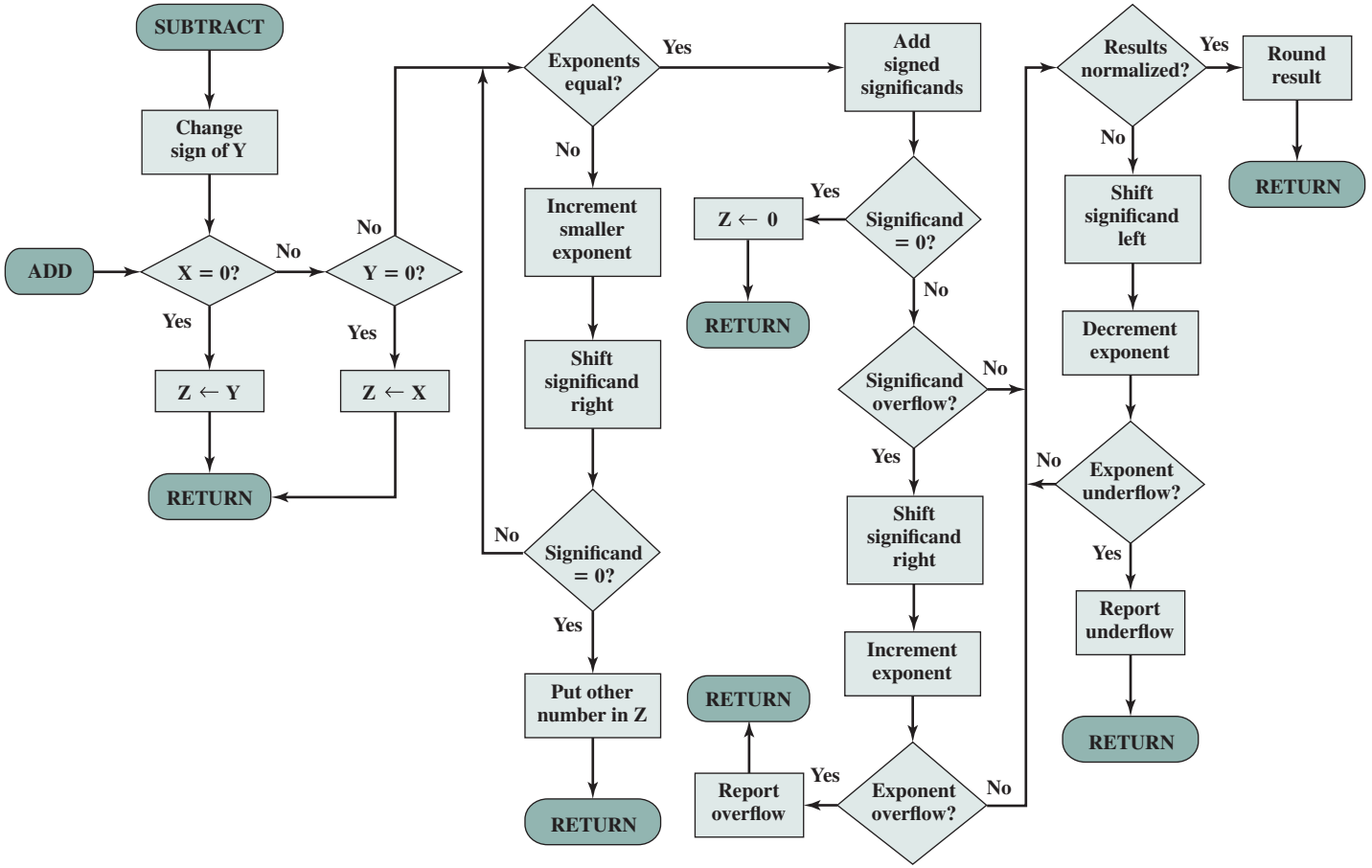


Figure 10.22 Floating-Point Addition and Subtraction ($Z \leftarrow X \pm Y$)

is achieved by repeatedly shifting the magnitude portion of the significand right 1 digit and incrementing the exponent until the two exponents are equal. (Note that if the implied base is 16, a shift of 1 digit is a shift of 4 bits.) If this process results in a 0 value for the significand, then the other number is reported as the result. Thus, if two numbers have exponents that differ significantly, the lesser number is lost.

Phase 3. Addition: Next, the two significands are added together, taking into account their signs. Because the signs may differ, the result may be 0. There is also the possibility of significand overflow by 1 digit. If so, the significand of the result is shifted right and the exponent is incremented. An exponent overflow could occur as a result; this would be reported and the operation halted.

Phase 4. Normalization: The final phase normalizes the result. Normalization consists of shifting significand digits left until the most significant digit (bit, or 4 bits for base-16 exponent) is nonzero. Each shift causes a decrement of the exponent and thus could cause an exponent underflow. Finally, the result must be rounded off and then reported. We defer a discussion of rounding until after a discussion of multiplication and division.

Multiplication and Division

Floating-point multiplication and division are much simpler processes than addition and subtraction, as the following discussion indicates.

We first consider multiplication, illustrated in Figure 10.23. First, if either operand is 0, 0 is reported as the result. The next step is to add the exponents. If the exponents are stored in biased form, the exponent sum would have doubled the bias. Thus, the bias value must be subtracted from the sum. The result could be either an exponent overflow or underflow, which would be reported, ending the algorithm.

If the exponent of the product is within the proper range, the next step is to multiply the significands, taking into account their signs. The multiplication is performed in the same way as for integers. In this case, we are dealing with a sign-magnitude representation, but the details are similar to those for twos complement representation. The product will be double the length of the multiplier and multiplicand. The extra bits will be lost during rounding.

After the product is calculated, the result is then normalized and rounded, as was done for addition and subtraction. Note that normalization could result in exponent underflow.

Finally, let us consider the flowchart for division depicted in Figure 10.24. Again, the first step is testing for 0. If the divisor is 0, an error report is issued, or the result is set to infinity, depending on the implementation. A dividend of 0 results in 0. Next, the divisor exponent is subtracted from the dividend exponent. This removes the bias, which must be added back in. Tests are then made for exponent underflow or overflow.

The next step is to divide the significands. This is followed with the usual normalization and rounding.

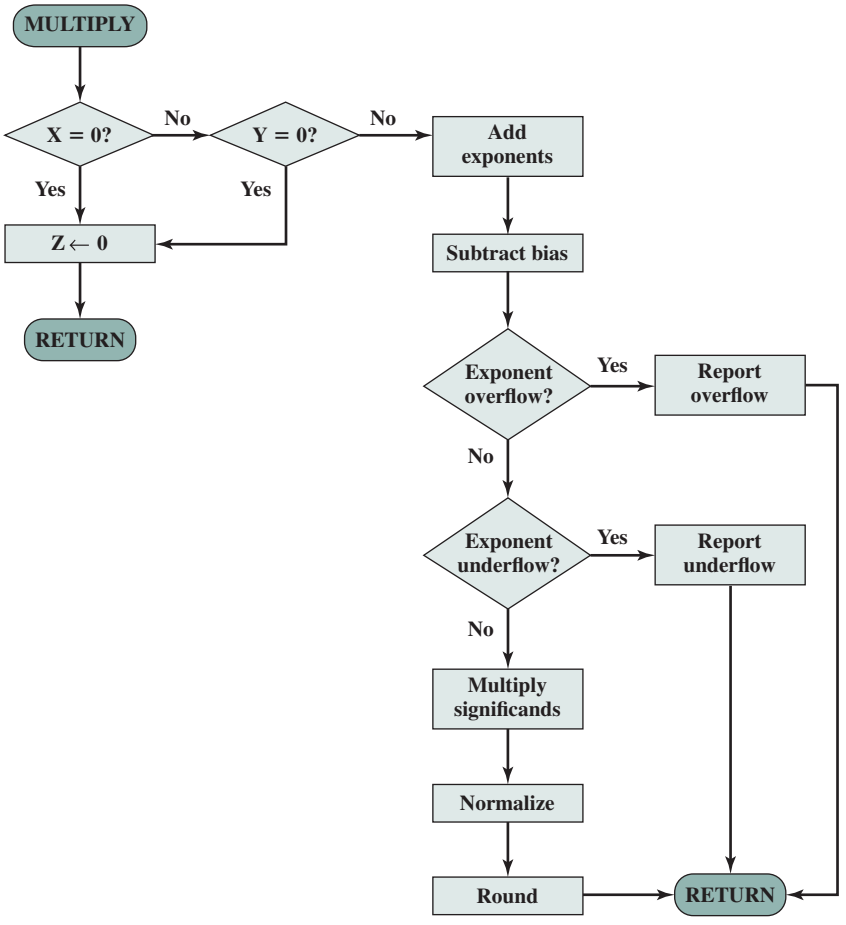


Figure 10.23 Floating-Point Multiplication ($Z \leftarrow X \pm Y$)

Precision Considerations

GUARD BITS We mentioned that, prior to a floating-point operation, the exponent and significand of each operand are loaded into ALU registers. In the case of the significand, the length of the register is almost always greater than the length of the significand plus an implied bit. The register contains additional bits, called guard bits, which are used to pad out the right end of the significand with 0s.

The reason for the use of guard bits is illustrated in Figure 10.25. Consider numbers in the IEEE format, which has a 24-bit significand, including an implied 1 bit to the left of the binary point. Two numbers that are very close in value are $x = 1.00 \cdots 00 \times 2^1$ and $y = 1.11 \cdots 11 \times 2^0$. If the smaller number is to be subtracted from the larger, it must be shifted right 1 bit to align the exponents. This is shown in Figure 10.25a. In the process, y loses 1 bit of significance; the result is 2^{-22}. The same operation is repeated in

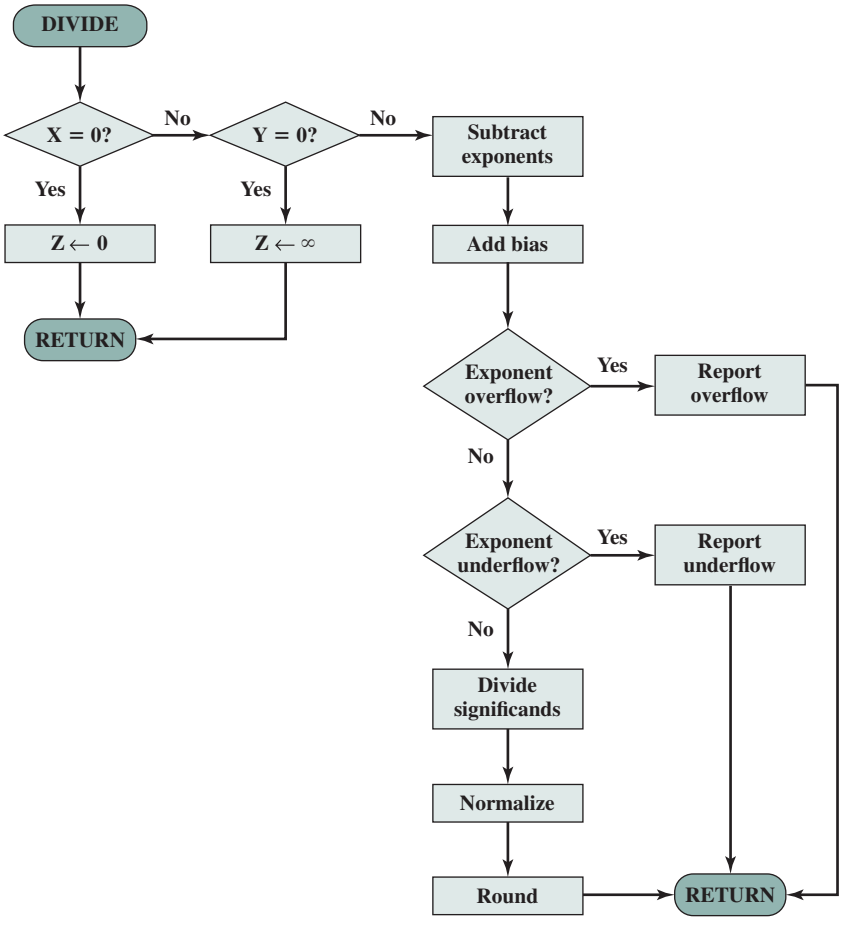


Figure 10.24 Floating-Point Division (Z ← X/Y)

$$
\begin{aligned}
x &= 1.000.....00 \times 2^{1} \\
-y &= 0.111.....11 \times 2^{1} \\
z &= 0.000.....01 \times 2^{1} \\
&= 1.000.....00 \times 2^{-22}
\end{aligned}
$$

(a) Binary example, without guard bits

$$
\begin{aligned}
x &= .100000 \times 16^{1} \\
-y &= .0FFFFF \times 16^{1} \\
z &= .000001 \times 16^{1} \\
&= .100000 \times 16^{-4}
\end{aligned}
$$

(c) Hexadecimal example, without guard bits

$$
\begin{aligned}
x &= 1.000.....00\ 0000 \times 2^{1} \\
-y &= 0.111.....11\ 1000 \times 2^{1} \\
z &= 0.000.....00\ 1000 \times 2^{1} \\
&= 1.000.....00\ 0000 \times 2^{-23}
\end{aligned}
$$

(b) Binary example, with guard bits

$$
\begin{aligned}
x &= .100000\ 00 \times 16^{1} \\
-y &= .0FFFFF\ F0 \times 16^{1} \\
z &= .000000\ 10 \times 16^{1} \\
&= .100000\ 00 \times 16^{-5}
\end{aligned}
$$

(d) Hexadecimal example, with guard bits

Figure 10.25 The Use of Guard Bits

part (b) with the addition of guard bits. Now the least significant bit is not lost due to alignment, and the result is 2^{-23}, a difference of a factor of 2 from the previous answer. When the radix is 16, the loss of precision can be greater. As Figures 10.25c and (d) show, the difference can be a factor of 16.

ROUNDING Another detail that affects the precision of the result is the rounding policy. The result of any operation on the significands is generally stored in a longer register. When the result is put back into the floating-point format, the extra bits must be eliminated in such a way as to produce a result that is close to the exact result. This process is called **rounding**.

A number of techniques have been explored for performing rounding. In fact, the IEEE standard lists four alternative approaches:

- **Round to nearest:** The result is rounded to the nearest representable number.
- **Round toward $+\infty$:** The result is rounded up toward plus infinity.
- **Round toward $-\infty$:** The result is rounded down toward negative infinity.
- **Round toward 0:** The result is rounded toward zero.

Let us consider each of these policies in turn. **Round to nearest** is the default rounding mode listed in the standard and is defined as follows: The representable value nearest to the infinitely precise result shall be delivered.

If the extra bits, beyond the 23 bits that can be stored, are 10010, then the extra bits amount to more than one-half of the last representable bit position. In this case, the correct answer is to add binary 1 to the last representable bit, rounding up to the next representable number. Now consider that the extra bits are 01111. In this case, the extra bits amount to less than one-half of the last representable bit position. The correct answer is simply to drop the extra bits (truncate), which has the effect of rounding down to the next representable number.

The standard also addresses the special case of extra bits of the form 10000.... Here the result is exactly halfway between the two possible representable values. One possible technique here would be to always truncate, as this would be the simplest operation. However, the difficulty with this simple approach is that it introduces a small but cumulative bias into a sequence of computations. What is required is an unbiased method of rounding. One possible approach would be to round up or down on the basis of a random number so that, on average, the result would be unbiased. The argument against this approach is that it does not produce predictable, deterministic results. The approach taken by the IEEE standard is to force the result to be even: If the result of a computation is exactly midway between two representable numbers, the value is rounded up if the last representable bit is currently 1 and not rounded up if it is currently 0.

The next two options, **rounding to plus** and **minus infinity**, are useful in implementing a technique known as interval arithmetic. Interval arithmetic provides an efficient method for monitoring and controlling errors in floating-point computations by producing two values for each result. The two values correspond to the lower and upper endpoints of an interval that contains the true result. The width of the interval, which is the difference between the upper and lower endpoints, indicates the accuracy of the result. If the endpoints of an interval are not representable, then the interval endpoints are rounded down and up, respectively. Although the width of the interval may vary according to implementation, many algorithms have been designed to produce narrow intervals. If the range between the upper and lower bounds is sufficiently narrow, then a sufficiently accurate result has been obtained. If not, at least we know this and can perform additional analysis.

The final technique specified in the standard is **round toward zero**. This is, in fact, simple truncation: The extra bits are ignored. This is certainly the simplest technique. However, the result is that the magnitude of the truncated value is always less than or equal to the more precise original value, introducing a consistent bias toward zero in the operation. This is a serious bias because it affects every operation for which there are nonzero extra bits.

IEEE Standard for Binary Floating-Point Arithmetic

IEEE 754 goes beyond the simple definition of a format to lay down specific practices and procedures so that floating-point arithmetic produces uniform, predictable results independent of the hardware platform. One aspect of this has already been discussed, namely rounding. This subsection looks at three other topics: infinity, NaNs, and subnormal numbers.

INFINITY Infinity arithmetic is treated as the limiting case of real arithmetic, with the infinity values given the following interpretation:

$$-\infty < (\text{every finite number}) < +\infty$$

With the exception of the special cases discussed subsequently, any arithmetic operation involving infinity yields the obvious result.

For example:

$$\begin{array}{ll} 5 + (+\infty) = +\infty & 5 \div (+\infty) = +0 \\ 5 - (+\infty) = -\infty & (+\infty) + (+\infty) = +\infty \\ 5 + (-\infty) = -\infty & (-\infty) + (-\infty) = -\infty \\ 5 - (-\infty) = +\infty & (-\infty) - (+\infty) = -\infty \\ 5 \times (+\infty) = +\infty & (+\infty) - (-\infty) = +\infty \end{array}$$

QUIET AND SIGNALING NANS A NaN is a symbolic entity encoded in floating-point format, of which there are two types: signaling and quiet. A signaling NaN signals an invalid operation exception whenever it appears as an operand. Signaling

Table 10.7 Operations that Produce a Quiet NaN

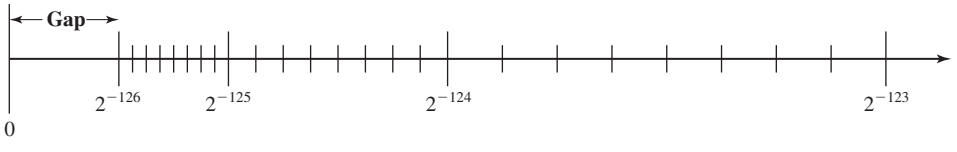
Operation	Quiet NaN Produced By
Any	Any operation on a signaling NaN
Add or subtract	Magnitude subtraction of infinities: $(+\infty) + (-\infty)$ $(-\infty) + (+\infty)$ $(+\infty) - (+\infty)$ $(-\infty) - (-\infty)$
Multiply	$0 \times \infty$
Division	$\frac{0}{0}$ or $\frac{\infty}{\infty}$
Remainder	x REM 0 or ∞ REM y
Square root	$\sqrt{x}$, where $x < 0$

NaNs afford values for uninitialized variables and arithmetic-like enhancements that are not the subject of the standard. A quiet NaN propagates through almost every arithmetic operation without signaling an exception. Table 10.7 indicates operations that will produce a quiet NaN.

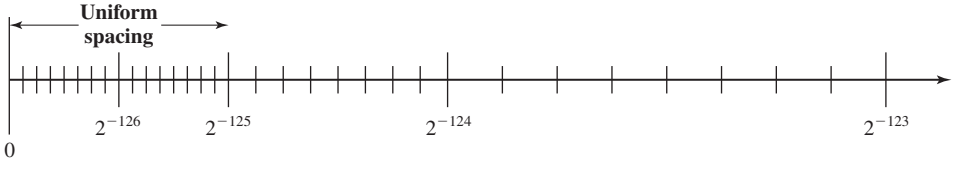
Note that both types of NaNs have the same general format (Table 10.4): an exponent of all ones and a nonzero fraction. The actual bit pattern of the nonzero fraction is implementation dependent; the fraction values can be used to distinguish quiet NaNs from signaling NaNs and to specify particular exception conditions.

SUBNORMAL NUMBERS Subnormal numbers are included in IEEE 754 to handle cases of exponent underflow. When the exponent of the result becomes too small (a negative exponent with too large a magnitude), the result is subnormalized by right shifting the fraction and incrementing the exponent for each shift until the exponent is within a representable range.

Figure 10.26 illustrates the effect of including subnormal numbers. The representable numbers can be grouped into intervals of the form $[2^n, 2^{n+1}]$. Within



(a) 32-bit format without subnormal numbers



(b) 32-bit format with subnormal numbers

Figure 10.26 The Effect of IEEE 754 Subnormal Numbers

each such interval, the exponent portion of the number remains constant while the fraction varies, producing a uniform spacing of representable numbers within the interval. As we get closer to zero, each successive interval is half the width of the preceding interval but contains the same number of representable numbers. Hence the density of representable numbers increases as we approach zero. However, if only normal numbers are used, there is a gap between the smallest normal number and 0. In the case of the 32-bit IEEE 754 format, there are 2^{23} representable numbers in each interval, and the smallest representable positive number is 2^{-126}. With the addition of subnormal numbers, an additional $2^{23} - 1$ numbers are uniformly added between 0 and 2^{-126}.

The use of subnormal numbers is referred to as *gradual underflow* [COON81]. Without subnormal numbers, the gap between the smallest representable nonzero number and zero is much wider than the gap between the smallest representable nonzero number and the next larger number. Gradual underflow fills in that gap and reduces the impact of exponent underflow to a level comparable with roundoff among the normal numbers.

10.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

arithmetic and logic unit (ALU)	minuend	radix point
arithmetic shift	multiplicand	range extension
base	multiplier	remainder
biased representation	negative overflow	rounding
dividend	negative underflow	sign bit
divisor	normal number	sign-magnitude representation
exponent	ones complement representation	significand
exponent overflow	overflow	significand overflow
exponent underflow	partial product	significand underflow
fixed-point representation	positive overflow	subnormal number
floating-point representation	positive underflow	subtrahend
guard bits	product	twos complement representation
mantissa	quotient	

Review Questions

10.1 Briefly explain the following representations: sign magnitude, twos complement, biased.

10.2 Explain how to determine if a number is negative in the following representations: sign magnitude, twos complement, biased.

10.3 What is the sign-extension rule for twos complement numbers?

10.4 How can you form the negation of an integer in twos complement representation?

10.5 In general terms, when does the twos complement operation on an n-bit integer produce the same integer?

10.6 What is the difference between the twos complement representation of a number and the twos complement of a number?

10.7 If we treat two twos complement numbers as unsigned integers for purposes of addition, the result is correct if interpreted as a twos complement number. This is not true for multiplication. Why?

10.8 What are the four essential elements of a number in floating-point notation?

10.9 What is the benefit of using biased representation for the exponent portion of a floating-point number?

10.10 What are the differences among positive overflow, exponent overflow, and significand overflow?

10.11 What are the basic elements of floating-point addition and subtraction?

10.12 Give a reason for the use of guard bits.

10.13 List four alternative methods of rounding the result of a floating-point operation.

Problems

10.1 Represent the following decimal numbers in both binary sign/magnitude and twos complement using 16 bits: $+512$; -29.

10.2 Represent the following twos complement values in decimal: 1101011; 0101101.

10.3 Another representation of binary integers that is sometimes encountered is **ones complement**. Positive integers are represented in the same way as sign magnitude. A negative integer is represented by taking the Boolean complement of each bit of the corresponding positive number.

a. Provide a definition of ones complement numbers using a weighted sum of bits, similar to Equations (10.1) and (10.2).

b. What is the range of numbers that can be represented in ones complement?

c. Define an algorithm for performing addition in ones complement arithmetic.

Note: Ones complement arithmetic disappeared from hardware in the 1960s, but still survives checksum calculations for the Internet Protocol (IP) and the Transmission Control Protocol (TCP).

10.4 Add columns to Table 10.1 for sign magnitude and ones complement.

10.5 Consider the following operation on a binary word. Start with the least significant bit. Copy all bits that are 0 until the first bit is reached and copy that bit, too. Then take the complement of each bit thereafter. What is the result?

10.6 In Section 10.3, the twos complement operation is defined as follows. To find the twos complement of X, take the Boolean complement of each bit of X, and then add 1.

a. Show that the following is an equivalent definition. For an n-bit integer X, the twos complement of X is formed by treating X as an unsigned integer and calculating $(2_n - X)$.

b. Demonstrate that Figure 10.5 can be used to support graphically the claim in part (a), by showing how a clockwise movement is used to achieve subtraction.

10.7 The r's complement of an n-digit number N in base r is defined as $r^n - N$ for $N \neq 0$ and 0 for $N = 0$. Find the tens complement of the decimal number 13,250.

10.8 Calculate $(72,530 - 13,250)$ using tens complement arithmetic. Assume rules similar to those for twos complement arithmetic.

10.9 Consider the twos complement addition of two n-bit numbers:

$$z_{n-1}z_{n-2} \cdots z_0 = x_{n-1}x_{n-2} \cdots x_0 + y_{n-1}y_{n-2} \cdots y_0$$

Assume that bitwise addition is performed with a carry bit c_i generated by the addition of x_i, y_i, and c_{i-1}. Let ν be a binary variable indicating overflow when $\nu = 1$. Fill in the values in the table.

Input	x_{n-1}	0	0	0	0	1	1	1	1
	y_{n-1}	0	0	1	1	0	0	1	1
	c_{n-2}	0	1	0	1	0	1	0	1
Output	z_{n-1}								
	v								

10.10 Assume numbers are represented in 8-bit twos complement representation. Show the calculation of the following:

a. $6 + 13$ **b.** $-6 + 13$ **c.** $6 - 13$ **d.** $-6 - 13$

10.11 Find the following differences using twos complement arithmetic:

a. $\begin{array}{r} 111000 \\ -110011 \end{array}$ **b.** $\begin{array}{r} 11001100 \\ -101110 \end{array}$ **c.** $\begin{array}{r} 111100001111 \\ -110011110011 \end{array}$ **d.** $\begin{array}{r} 11000011 \\ -11101000 \end{array}$

10.12 Is the following a valid alternative definition of overflow in twos complement arithmetic?

If the exclusive-OR of the carry bits into and out of the leftmost column is 1, then there is an overflow condition. Otherwise, there is not.

10.13 Compare Figures 10.9 and 10.12. Why is the C bit not used in the latter?

10.14 Given $x = 0101$ and $y = 1010$ in twos complement notation (i.e., $x = 5, y = -6$), compute the product $p = x \times y$ with Booth's algorithm.

10.15 Use the Booth algorithm to multiply 23 (multiplicand) by 29 (multiplier), where each number is represented using 6 bits.

10.16 Prove that the multiplication of two n-digit numbers in base B gives a product of no more than $2n$ digits.

10.17 Verify the validity of the unsigned binary division algorithm of Figure 10.16 by showing the steps involved in calculating the division depicted in Figure 10.15. Use a presentation similar to that of Figure 10.17.

10.18 The twos complement integer division algorithm described in Section 10.3 is known as the restoring method because the value in the A register must be restored following unsuccessful subtraction. A slightly more complex approach, known as nonrestoring, avoids the unnecessary subtraction and addition. Propose an algorithm for this latter approach.

10.19 Under computer integer arithmetic, the quotient J/K of two integers J and K is less than or equal to the usual quotient. True or false?

10.20 Divide -145 by 13 in binary twos complement notation, using 12-bit words. Use the algorithm described in Section 10.3.

10.21 **a.** Consider a fixed-point representation using decimal digits, in which the implied radix point can be in any position (to the right of the least significant digit, to the right of the most significant digit, and so on). How many decimal digits are needed to represent the approximations of both Planck's constant (6.63×10^{-27}) and Avogadro's number (6.02×10^{23})? The implied radix point must be in the same position for both numbers.

b. Now consider a decimal floating-point format with the exponent stored in a biased representation with a bias of 50. A normalized representation is assumed. How many decimal digits are needed to represent these constants in this floating-point format?

10.22 Assume that the exponent e is constrained to lie in the range $0 \leq e \leq X$, with a bias of q, that the base is b, and that the significand is p digits in length.

a. What are the largest and smallest positive values that can be written?

b. What are the largest and smallest positive values that can be written as normalized floating-point numbers?

10.23 Express the following numbers in IEEE 32-bit floating-point format:

a. −5 **b.** −6 **c.** −1.5 **d.** 384 **e.** 1/16 **f.** −1/32

10.24 The following numbers use the IEEE 32-bit floating-point format. What is the equivalent decimal value?

a. 1 10000011 11000000000000000000000
b. 0 01111110 10100000000000000000000
c. 0 10000000 00000000000000000000000

10.25 Consider a reduced 7-bit IEEE floating-point format, with 3 bits for the exponent and 3 bits for the significand. List all 127 values.

10.26 Express the following numbers in IBM's 32-bit floating-point format, which uses a 7-bit exponent with an implied base of 16 and an exponent bias of 64 (40 hexadecimal). A normalized floating-point number requires that the leftmost hexadecimal digit be nonzero; the implied radix point is to the left of that digit.

a. 1.0	**c.** 1/64	**e.** − 15.0	**g.** 7.2×10^{75}
b. 0.5	**d.** 0.0	**f.** 5.4×10^{-79}	**h.** 65,535

10.27 Let 5BCA0000 be a floating-point number in IBM format, expressed in hexadecimal. What is the decimal value of the number?

10.28 What would be the bias value for

a. A base-2 exponent (B = 2) in a 6-bit field?
b. A base-8 exponent (B = 8) in a 7-bit field?

10.29 Draw a number line similar to that in Figure 10.19b for the floating-point format of Figure 10.21b.

10.30 Consider a floating-point format with 8 bits for the biased exponent and 23 bits for the significand. Show the bit pattern for the following numbers in this format:

a. −720 **b.** 0.645

10.31 The text mentions that a 32-bit format can represent a maximum of 2^{32} different numbers. How many different numbers can be represented in the IEEE 32-bit format? Explain.

10.32 Any floating-point representation used in a computer can represent only certain real numbers exactly; all others must be approximated. If A' is the stored value approximating the real value A, then the relative error, r, is expressed as

$$r = \frac{A - A'}{A}$$

Represent the decimal quantity + 0.4 in the following floating-point format: base = 2; exponent: biased, 4 bits; significand, 7 bits. What is the relative error?

10.33 If $A = 1.427$, find the relative error if A is truncated to 1.42 and if it is rounded to 1.43.

10.34 When people speak about inaccuracy in floating-point arithmetic, they often ascribe errors to cancellation that occurs during the subtraction of nearly equal quantities. But when X and Y are approximately equal, the difference $X - Y$ is obtained exactly, with no error. What do these people really mean?

10.35 Numerical values A and B are stored in the computer as approximations A' and B' Neglecting any further truncation or roundoff errors, show that the relative error of the product is approximately the sum of the relative errors in the factors.

10.36 One of the most serious errors in computer calculations occurs when two nearly equal numbers are subtracted. Consider $A = 0.22288$ and $B = 0.22211$. The computer truncates all values to four decimal digits. Thus $A' = 0.2228$ and $B' = 0.2221$.

a. What are the relative errors for A' and B'?
b. What is the relative error for $C' = A' - B'$?

10.37 To get some feel for the effects of denormalization and gradual underflow, consider a decimal system that provides 6 decimal digits for the significand and for which the smallest normalized number is 10^{-99}. A normalized number has one nonzero decimal digit to the left of the decimal point. Perform the following calculations and denormalize the results. Comment on the results.

a. $(2.50000 \times 10^{-60}) \times (3.50000 \times 10^{-43})$
b. $(2.50000 \times 10^{-60}) \times (3.50000 \times 10^{-60})$
c. $(5.67834 \times 10^{-97}) - (5.67812 \times 10^{-97})$

10.38 Show how the following floating-point additions are performed (where significands are truncated to 4 decimal digits). Show the results in normalized form.

a. $5.566 \times 10^{2} + 7.777 \times 10^{2}$
b. $3.344 \times 10^{1} + 8.877 \times 10^{-2}$

10.39 Show how the following floating-point subtractions are performed (where significands are truncated to 4 decimal digits). Show the results in normalized form.

a. $7.744 \times 10^{-3} - 6.666 \times 10^{-3}$ **b.** $8.844 \times 10^{-3} - 2.233 \times 10^{-1}$

10.40 Show how the following floating-point calculations are performed (where significands are truncated to 4 decimal digits). Show the results in normalized form.

a. $(2.255 \times 10^{1}) \times (1.234 \times 10^{0})$ **b.** $(8.833 \times 10^{2}) \div (5.555 \times 10^{4})$

CHAPTER 11

DIGITAL LOGIC

11.1 Boolean Algebra

11.2 Gates

11.3 Combinational Circuits
- Implementation of Boolean Functions
- Multiplexers
- Decoders
- Read-Only Memory
- Adders

11.4 Sequential Circuits
- Flip-Flops
- Registers
- Counters

11.5 Programmable Logic Devices
- Programmable Logic Array
- Field-Programmable Gate Array

11.6 Key Terms and Problems

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Understand the basic operations of **Boolean algebra**.
- Distinguish among the different types of **flip-flops**.
- Use a Karnaugh map to simplify a Boolean expression.
- Present an overview of **programmable logic devices**.

The operation of the digital computer is based on the storage and processing of binary data. Throughout this book, we have assumed the existence of storage elements that can exist in one of two stable states, and of circuits than can operate on binary data under the control of control signals to implement the various computer functions. In this chapter, we suggest how these storage elements and circuits can be implemented in digital logic, specifically with combinational and sequential circuits. The chapter begins with a brief review of Boolean algebra, which is the mathematical foundation of digital logic. Next, the concept of a gate is introduced. Finally, combinational and sequential circuits, which are constructed from **gates**, are described.

11.1 BOOLEAN ALGEBRA

The digital circuitry in digital computers and other digital systems is designed, and its behavior is analyzed, with the use of a mathematical discipline known as **Boolean algebra**. The name is in honor of an English mathematician George Boole, who proposed the basic principles of this algebra in 1854 in his treatise, *An Investigation of the Laws of Thought on Which to Found the Mathematical Theories of Logic and Probabilities.* In 1938, Claude Shannon, a research assistant in the Electrical Engineering Department at M.I.T., suggested that Boolean algebra could be used to solve problems in relay-switching circuit design [SHAN38].[1] Shannon's techniques were subsequently used in the analysis and design of electronic digital circuits. Boolean algebra turns out to be a convenient tool in two areas:

- **Analysis:** It is an economical way of describing the function of digital circuitry.
- **Design:** Given a desired function, Boolean algebra can be applied to develop a simplified implementation of that function.

As with any algebra, Boolean algebra makes use of variables and operations. In this case, the variables and operations are logical variables and operations. Thus, a variable may take on the value 1 (TRUE) or 0 (FALSE). The basic logical

[1]The paper is available at box.com/COA10e.

operations are AND, OR, and NOT, which are symbolically represented by dot, plus sign, and overbar:[2]

$$\text{A AND B} = \text{A} \cdot \text{B}$$
$$\text{A OR B} = \text{A} + \text{B}$$
$$\text{NOT A} = \overline{\text{A}}$$

The operation AND yields true (binary value 1) if and only if both of its operands are true. The operation OR yields true if either or both of its operands are true. The unary operation NOT inverts the value of its operand. For example, consider the equation

$$\text{D} = \text{A} + (\overline{\text{B}} \cdot \text{C})$$

D is equal to 1 if A is 1 or if both B = 0 and C = 1. Otherwise D is equal to 0.

Several points concerning the notation are needed. In the absence of parentheses, the AND operation takes precedence over the OR operation. Also, when no ambiguity will occur, the AND operation is represented by simple concatenation instead of the dot operator. Thus,

$$\text{A} + \text{B} \cdot \text{C} = \text{A} + (\text{B} \cdot \text{C}) = \text{A} + \text{BC}$$

all mean: Take the AND of B and C; then take the OR of the result and A.

Table 11.1a defines the basic logical operations in a form known as a *truth table,* which lists the value of an operation for every possible combination of values of operands. The table also lists three other useful operators: XOR, **NAND**, and **NOR**. The exclusive-or (XOR) of two logical operands is 1 if and only if exactly one of the operands has the value 1. The NAND function is the complement (NOT) of the AND function, and the NOR is the complement of OR:

$$\text{A NAND B} = \text{NOT (A AND B)} = \overline{\text{AB}}$$
$$\text{A NOR B} = \text{NOT (A OR B)} = \overline{\text{A} + \text{B}}$$

As we shall see, these three new operations can be useful in implementing certain digital circuits.

The logical operations, with the exception of NOT, can be generalized to more than two variables, as shown in Table 11.1b.

Table 11.2 summarizes key identities of Boolean algebra. The equations have been arranged in two columns to show the complementary, or dual, nature of the AND and OR operations. There are two classes of identities: basic rules (or *postulates*), which are stated without proof, and other identities that can be derived from the basic postulates. The postulates define the way in which Boolean expressions are interpreted. One of the two distributive laws is worth noting because it differs from what we would find in ordinary algebra:

$$\text{A} + (\text{B} \cdot \text{C}) = (\text{A} + \text{B}) \cdot (\text{A} + \text{C})$$

[2]Logical NOT is often indicated by an apostrophe: NOT A = A′.

Table 11.1 Boolean Operators

(a) Boolean Operators of Two Input Variables

P	Q	NOT P ($\overline{P}$)	P AND Q ($P \cdot Q$)	P OR Q ($P + Q$)	P NAND Q ($\overline{P \cdot Q}$)	P NOR Q ($\overline{P + Q}$)	P XOR Q ($P \oplus Q$)
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

(b) Boolean Operators Extended to More than Two Inputs (A, B, . . .)

Operation	Expression	Output = 1 if
AND	$A \cdot B \cdot \ldots$	All of the set {A, B, ...} are 1.
OR	$A + B + \ldots$	Any of the set {A, B, ...} are 1.
NAND	$\overline{A \cdot B \cdot \ldots}$	Any of the set {A, B, ...} are 0.
NOR	$\overline{A + B + \ldots}$	All of the set {A, B, ...} are 0.
XOR	$A \oplus B \oplus \ldots$	The set {A, B, ...} contains an odd number of ones.

The two bottommost expressions are referred to as DeMorgan's theorem. We can restate them as follows:

$$A \text{ NOR } B = \overline{A} \text{ AND } \overline{B}$$
$$A \text{ NAND } B = \overline{A} \text{ OR } \overline{B}$$

The reader is invited to verify the expressions in Table 11.2 by substituting actual values (1s and 0s) for the variables A, B, and C.

Table 11.2 Basic Identities of Boolean Algebra

Basic Postulates		
$A \cdot B = B \cdot A$	$A + B = B + A$	Commutative Laws
$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$	Distributive Laws
$1 \cdot A = A$	$0 + A = A$	Identity Elements
$A \cdot \overline{A} = 0$	$A + \overline{A} = 1$	Inverse Elements
Other Identities		
$0 \cdot A = 0$	$1 + A = 1$	
$A \cdot A = A$	$A + A = A$	
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$	$A + (B + C) = (A + B) + C$	Associative Laws
$\overline{A \cdot B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \cdot \overline{B}$	DeMorgan's Theorem

11.2 GATES

The fundamental building block of all digital logic circuits is the gate. Logical functions are implemented by the interconnection of gates.

A gate is an electronic circuit that produces an output signal that is a simple Boolean operation on its input signals. The basic gates used in digital logic are AND, OR, NOT, NAND, NOR, and XOR. Figure 11.1 depicts these six gates. Each gate is defined in three ways: graphic symbol, algebraic notation, and truth table. The symbology used in this chapter is from the IEEE standard, IEEE Std 91. Note that the inversion (NOT) operation is indicated by a circle.

Each gate shown in Figure 11.1 has one or two inputs and one output. However, as indicated in Table 11.1b, all of the gates except NOT can have more than two inputs. Thus, (X + Y + Z) can be implemented with a single **OR gate** with three inputs. When one or more of the values at the input are changed, the correct output signal appears almost instantaneously, delayed only by the propagation time of signals through the gate (known as the *gate delay*). The significance of this delay is discussed in Section 11.3. In some cases, a gate is implemented with two outputs, one output being the negation of the other output.

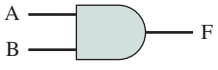
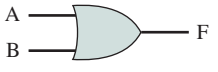
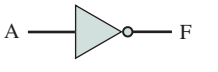
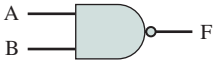
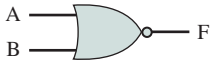
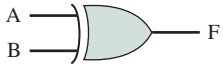
Name	Graphical Symbol	Algebraic Function	Truth Table
AND		$F = A \bullet B$ or $F = AB$	A B | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = A + B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT		$F = \overline{A}$ or $F = A'$	A | F 0 | 1 1 | 0
NAND		$F = \overline{AB}$	A B | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = \overline{A + B}$	A B | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
XOR		$F = A \oplus B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0

Figure 11.1 Basic Logic Gates

Here we introduce a common term: we say that to **assert** a signal is to cause a signal line to make a transition from its logically false (0) state to its logically true (1) state. The true (1) state is either a high or low voltage state, depending on the type of electronic circuitry.

Typically, not all gate types are used in implementation. Design and fabrication are simpler if only one or two types of gates are used. Thus, it is important to identify *functionally complete* sets of gates. This means that any Boolean function can be implemented using only the gates in the set. The following are functionally complete sets:

- AND, OR, NOT
- AND, NOT
- OR, NOT
- NAND
- NOR

It should be clear that AND, OR, and NOT gates constitute a functionally complete set, because they represent the three operations of Boolean algebra. For the AND and NOT gates to form a functionally complete set, there must be a way to synthesize the OR operation from the AND and NOT operations. This can be done by applying DeMorgan's theorem:

$$A + B = \overline{\overline{A} \cdot \overline{B}}$$

$$A \text{ OR } B = \text{NOT}((\text{NOT } A)\text{AND } (\text{NOT } B))$$

Similarly, the OR and NOT operations are functionally complete because they can be used to synthesize the AND operation.

Figure 11.2 shows how the AND, OR, and NOT functions can be implemented solely with NAND gates, and Figure 11.3 shows the same thing for NOR gates. For this reason, digital circuits can be, and frequently are, implemented solely with NAND gates or solely with NOR gates.

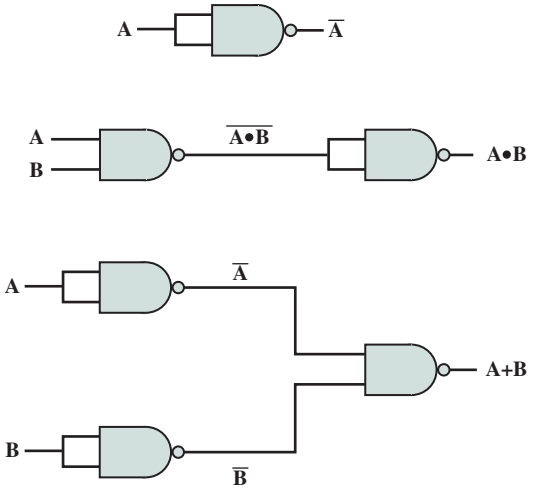


Figure 11.2 Some Uses of NAND Gates

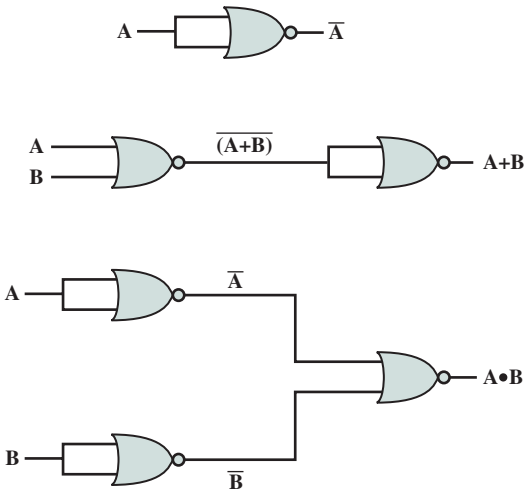


Figure 11.3 Some Uses of NOR Gates

With gates, we have reached the most primitive circuit level of computer hardware. An examination of the transistor combinations used to construct gates departs from that realm and enters the realm of electrical engineering. For our purposes, however, we are content to describe how gates can be used as building blocks to implement the essential logical circuits of a digital computer.

11.3 COMBINATIONAL CIRCUITS

A **combinational circuit** is an interconnected set of gates whose output at any time is a function only of the input at that time. As with a single gate, the appearance of the input is followed almost immediately by the appearance of the output, with only gate delays.

In general terms, a combinational circuit consists of n binary inputs and m binary outputs. As with a gate, a combinational circuit can be defined in three ways:

- **Truth table:** For each of the 2^n possible combinations of input signals, the binary value of each of the m output signals is listed.
- **Graphical symbols:** The interconnected layout of gates is depicted.
- **Boolean equations:** Each output signal is expressed as a Boolean function of its input signals.

Implementation of Boolean Functions

Any Boolean function can be implemented in electronic form as a network of gates. For any given function, there are a number of alternative realizations. Consider the Boolean function represented by the truth table in Table 11.3. We can express this function by simply itemizing the combinations of values of A, B, and C that cause F to be 1:

$$F + \overline{A}B\overline{C} + \overline{A}BC + AB\overline{C} \qquad \textbf{(11.1)}$$

Table 11.3 A Boolean Function of Three Variables

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

There are three combinations of input values that cause F to be 1, and if any one of these combinations occurs, the result is 1. This form of expression, for self-evident reasons, is known as the **sum of products (SOP)** form. Figure 11.4 shows a straightforward implementation with AND, OR, and NOT gates.

Another form can also be derived from the truth table. The SOP form expresses that the output is 1 if any of the input combinations that produce 1 is true. We can also say that the output is 1 if none of the input combinations that produce 0 is true. Thus,

$$F = \overline{(\overline{A}\,\overline{B}\,\overline{C})} \cdot \overline{(\overline{A}\,\overline{B}\,C)} \cdot \overline{(A\,\overline{B}\,\overline{C})} \cdot \overline{(A\,\overline{B}\,C)} \cdot \overline{(A\,B\,C)}$$

This can be rewritten using a generalization of DeMorgan's theorem:

$$\overline{(X \cdot Y \cdot Z)} = \overline{X} + \overline{Y} + \overline{Z}$$

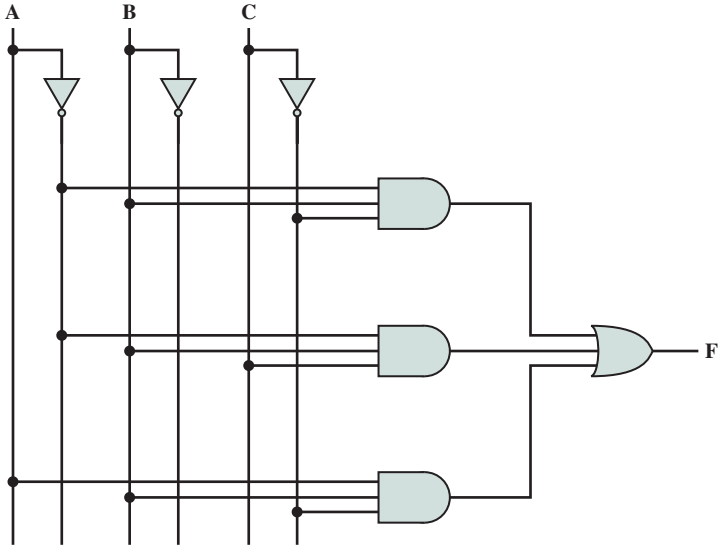


Figure 11.4 Sum-of-Products Implementation of Table 11.3

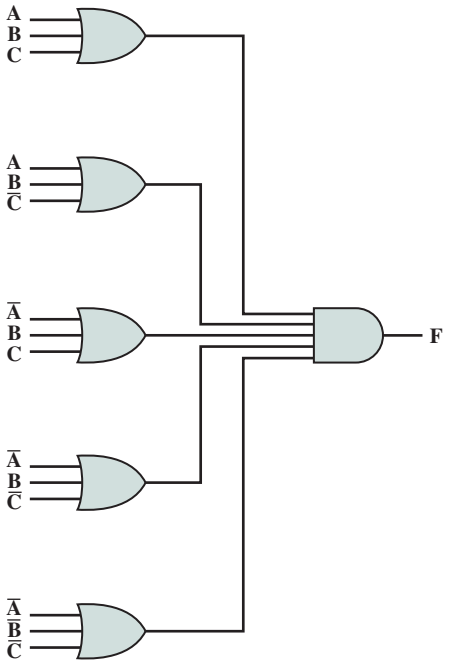


Figure 11.5 Product-of-Sums Implementation of Table 11.3

Thus,

$$F = (\overline{\overline{A}} + \overline{\overline{B}} + \overline{\overline{C}}) \cdot (\overline{\overline{A}} + \overline{\overline{B}} + \overline{C}) \cdot (\overline{\overline{A}} + \overline{\overline{B}} + \overline{C}) \cdot (\overline{\overline{A}} + \overline{\overline{B}} + \overline{C}) \cdot (\overline{A} + \overline{B} + \overline{C}) \quad \textbf{(11.2)}$$
$$= (A + B + C) \cdot (A + B + \overline{C}) \cdot (\overline{A} + B + C) \cdot (\overline{A} + B + \overline{C}) \cdot (\overline{A} + \overline{B} + \overline{C})$$

This is in the **product of sums (POS)** form, which is illustrated in Figure 11.5. For clarity, NOT gates are not shown. Rather, it is assumed that each input signal and its complement are available. This simplifies the logic diagram and makes the inputs to the gates more readily apparent.

Thus, a Boolean function can be realized in either SOP or POS form. At this point, it would seem that the choice would depend on whether the truth table contains more 1s or 0s for the output function: The SOP has one term for each 1, and the POS has one term for each 0. However, there are other considerations:

- It is often possible to derive a simpler Boolean expression from the truth table than either SOP or POS.
- It may be preferable to implement the function with a single gate type (NAND or NOR).

The significance of the first point is that, with a simpler Boolean expression, fewer gates will be needed to implement the function. Three methods that can be used to achieve simplification are

- Algebraic simplification
- Karnaugh maps
- Quine–McCluskey tables

ALGEBRAIC SIMPLIFICATION Algebraic simplification involves the application of the identities of Table 11.2 to reduce the Boolean expression to one with fewer elements. For example, consider again Equation (11.1). Some thought should convince the reader that an equivalent expression is

$$F = \overline{A}B + B\overline{C} \tag{11.3}$$

Or, even simpler,

$$F = B(\overline{A} + \overline{C})$$

This expression can be implemented as shown in Figure 11.6. The simplification of Equation (11.1) was done essentially by observation. For more complex expressions, some more systematic approach is needed.

KARNAUGH MAPS For purposes of simplification, the **Karnaugh map** is a convenient way of representing a Boolean function of a small number (up to four) of variables. The map is an array of 2^n squares, representing all possible combinations of values of n binary variables. Figure 11.7a shows the map of four squares for a function of two variables. It is essential for later purposes to list the combinations in the order 00, 01, 11, 10. Because the squares corresponding to the combinations are to be used for recording information, the combinations are customarily written above the squares. In the case of three variables, the representation is an arrangement of eight squares (Figure 11.7b), with the values for one of the variables to the left and for the other two variables above the squares. For four variables, 16 squares are needed, with the arrangement indicated in Figure 11.7c.

The map can be used to represent any Boolean function in the following way. Each square corresponds to a unique product in the sum-of-products form, with a 1 value corresponding to the variable and a 0 value corresponding to the NOT of that variable. Thus, the product $A\overline{B}$ corresponds to the fourth square in Figure 11.7a. For each such product in the function, 1 is placed in the corresponding square. Thus, for the two-variable example, the map corresponds to $A\overline{B} + \overline{A}B$. Given the truth table of a Boolean function, it is an easy matter to construct the map: for each combination of values of variables that produce a result of 1 in the truth table, fill in the corresponding square of the map with 1. Figure 11.7b shows the result for the truth table of Table 11.3. To convert from a Boolean expression to a map, it is first necessary to put the expression into what is referred to as *canonical* form: each term in the expression must contain each variable. So, for example, if we have Equation (11.3), we must first expand it into the full form of Equation (11.1) and then convert this to a map.

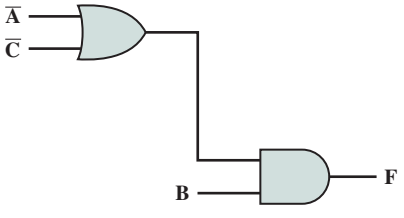


Figure 11.6 Simplified Implementation of Table A.3

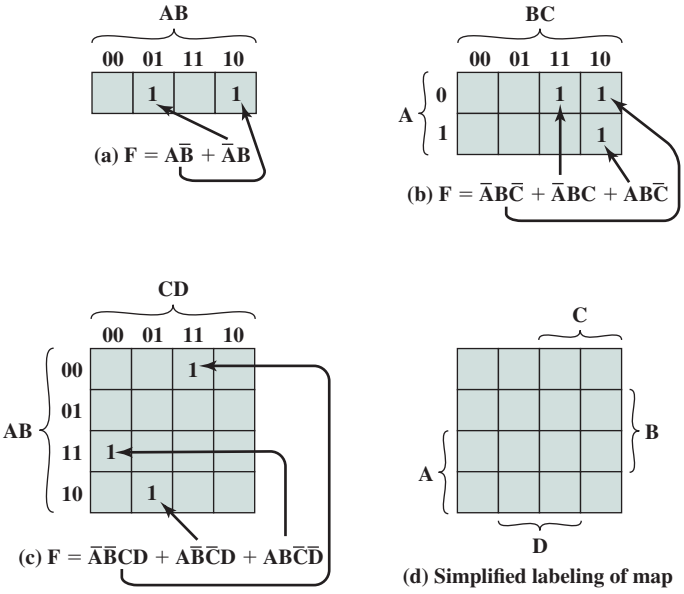


Figure 11.7 The Use of Karnaugh Maps to Represent Boolean Functions

The labeling used in Figure 11.7d emphasizes the relationship between variables and the rows and columns of the map. Here the two rows embraced by the symbol A are those in which the variable A has the value 1; the rows not embraced by the symbol A are those in which A is 0; similarly for B, C, and D.

Once the map of a function is created, we can often write a simple algebraic expression for it by noting the arrangement of the 1s on the map. The principle is as follows. Any two squares that are adjacent differ in only one of the variables. If two adjacent squares both have an entry of one, then the corresponding product terms differ in only one variable. In such a case, the two terms can be merged by eliminating that variable. For example, in Figure 11.8a, the two adjacent squares correspond to the two terms $\overline{A}B\overline{C}D$ and $\overline{A}BCD$. Thus, the function expressed is

$$\overline{A}B\overline{C}D + \overline{A}BCD = \overline{A}BD$$

This process can be extended in several ways. First, the concept of adjacency can be extended to include wrapping around the edge of the map. Thus, the top square of a column is adjacent to the bottom square, and the leftmost square of a row is adjacent to the rightmost square. These conditions are illustrated in Figures 11.8b and c. Second, we can group not just 2 squares but 2^n adjacent squares (i.e., 2, 4, 8, etc.). The next three examples in Figure 11.8 show groupings of 4 squares. Note that in this case, two of the variables can be eliminated. The last three examples show groupings of 8 squares, which allow three variables to be eliminated.

We can summarize the rules for simplification as follows:

1. Among the marked squares (squares with a 1), find those that belong to a unique largest block of 1, 2, 4, or 8 and circle those blocks.

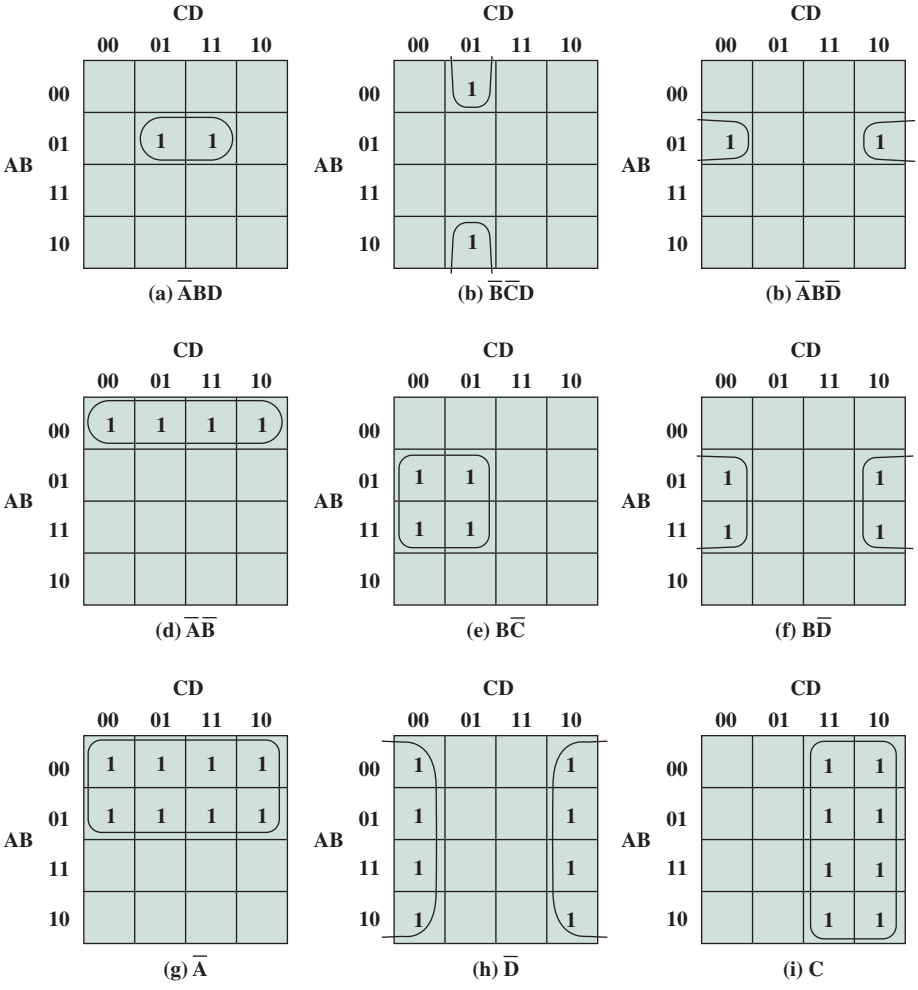


Figure 11.8 The Use of Karnaugh Maps

2. Select additional blocks of marked squares that are as large as possible and as few in number as possible, but include every marked square at least once. The results may not be unique in some cases. For example, if a marked square combines with exactly two other squares, and there is no fourth marked square to complete a larger group, then there is a choice to be made as to which of the two groupings to choose. When you are circling groups, you are allowed to use the same 1 value more than once.
3. Continue to draw loops around single marked squares, or pairs of adjacent marked squares, or groups of four, eight, and so on in such a way that every marked square belongs to at least one loop; then use as few of these blocks as possible to include all marked squares.

Figure 11.9a, based on Table 11.3, illustrates the simplification process. If any isolated 1s remain after the groupings, then each of these is circled as a group of 1s.

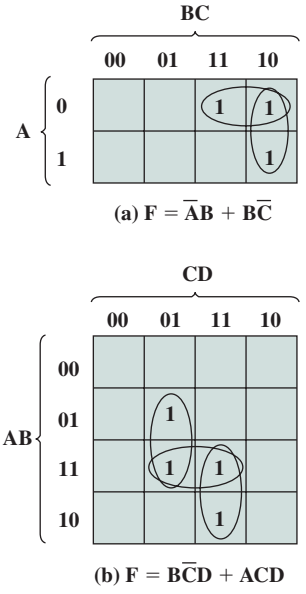


Figure 11.9 Overlapping Groups

Finally, before going from the map to a simplified Boolean expression, any group of 1s that is completely overlapped by other groups can be eliminated. This is shown in Figure 11.9b. In this case, the horizontal group is redundant and may be ignored in creating the Boolean expression.

One additional feature of Karnaugh maps needs to be mentioned. In some cases, certain combinations of values of variables never occur, and therefore the corresponding output never occurs. These are referred to as "don't care" conditions. For each such condition, the letter "d" is entered into the corresponding square of the map. In doing the grouping and simplification, each "d" can be treated as a 1 or 0, whichever leads to the simplest expression.

An example, presented in [HAYE98], illustrates the points we have been discussing. We would like to develop the Boolean expressions for a circuit that adds 1 to a packed decimal digit. For packed decimal, each decimal digit is represented by a 4-bit code, in the obvious way. Thus, $0 = 0000, 1 = 0001, \ldots, 8 = 1000$, and $9 = 1001$. The remaining 4-bit values, from 1010 to 1111, are not used. This code is also referred to as **Binary Coded Decimal (BCD)**.

Table 11.4 shows the truth table for producing a 4-bit result that is one more than a 4-bit BCD input. The addition is modulo 10. Thus, $9 + 1 = 0$. Also, note that six of the input codes produce "don't care" results, because those are not valid BCD inputs. Figure 11.10 shows the resulting Karnaugh maps for each of the output variables. The d squares are used to achieve the best possible groupings.

THE QUINE–MCCLUSKEY METHOD For more than four variables, the Karnaugh map method becomes increasingly cumbersome. With five variables, two 16×16 maps are needed, with one map considered to be on top of the other in three dimensions to achieve adjacency. Six variables require the use of four 16×16

Table 11.4 Truth Table for the One-Digit Packed Decimal Incrementer

	Input					Output			
Number	A	B	C	D	Number	W	X	Y	Z
0	0	0	0	0	1	0	0	0	1
1	0	0	0	1	2	0	0	1	0
2	0	0	1	0	3	0	0	1	1
3	0	0	1	1	4	0	1	0	0
4	0	1	0	0	5	0	1	0	1
5	0	1	0	1	6	0	1	1	0
6	0	1	1	0	7	0	1	1	1
7	0	1	1	1	8	1	0	0	0
8	1	0	0	0	9	1	0	0	1
9	1	0	0	1	0	0	0	0	0
Don't care condition	1	0	1	0		d	d	d	d
	1	0	1	1		d	d	d	d
	1	1	0	0		d	d	d	d
	1	1	0	1		d	d	d	d
	1	1	1	0		d	d	d	d
	1	1	1	1		d	d	d	D

tables in four dimensions! An alternative approach is a tabular technique, referred to as the Quine–McCluskey method. The method is suitable for programming on a computer to give an automatic tool for producing minimized Boolean expressions.

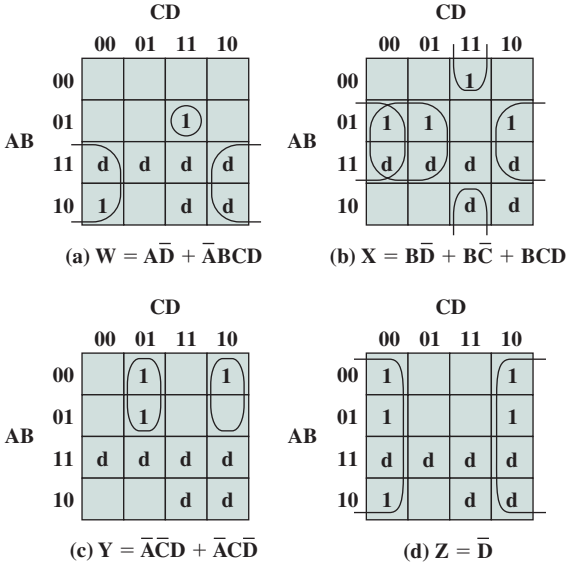


Figure 11.10 Karnaugh Maps for the Incrementer

The method is best explained by means of an example. Consider the following expression:

$$ABCD + AB\overline{C}D + AB\overline{C}\,\overline{D} + A\overline{B}CD + \overline{A}BCD + \overline{A}BC\overline{D} + \overline{A}B\overline{C}D + \overline{A}\,\overline{B}\,\overline{C}D$$

Let us assume that this expression was derived from a truth table. We would like to produce a minimal expression suitable for implementation with gates.

The first step is to construct a table in which each row corresponds to one of the product terms of the expression. The terms are grouped according to the number of complemented variables. That is, we start with the term with no complements, if it exists, then all terms with one complement, and so on. Table 11.5 shows the list for our example expression, with horizontal lines used to indicate the grouping. For clarity, each term is represented by a 1 for each uncomplemented variable and a 0 for each complemented variable. Thus, we group terms according to the number of 1s they contain. The index column is simply the decimal equivalent and is useful in what follows.

The next step is to find all pairs of terms that differ in only one variable, that is, all pairs of terms that are the same except that one variable is 0 in one of the terms and 1 in the other. Because of the way in which we have grouped the terms, we can do this by starting with the first group and comparing each term of the first group with every term of the second group. Then compare each term of the second group with all of the terms of the third group, and so on. Whenever a match is found, place a check next to each term, combine the pair by eliminating the variable that differs in the two terms, and add that to a new list. Thus, for example, the terms $\overline{A}BC\overline{D}$ and $\overline{A}BCD$ are combined to produce ABC. This process continues until the entire original table has been examined. The result is a new table with the following entries:

$\overline{A}\,\overline{C}D$	$AB\overline{C}$	$ABD_{\surd}$
	$B\overline{C}D_{\surd}$	ACD
	$\overline{A}BC$	$BCD_{\surd}$
	$\overline{A}BD_{\surd}$	

Table 11.5 First Stage of Quine–McCluskey Method
(for $F = ABCD + AB\overline{C}D + AB\overline{C}\,\overline{D} + A\overline{B}CD + \overline{A}BCD + \overline{A}BC\overline{D} + \overline{A}B\overline{C}D + \overline{A}\,\overline{B}\,\overline{C}D$)

Product Term	Index	A	B	C	D	
$\overline{A}\,\overline{B}\,\overline{C}D$	1	0	0	0	1	✓
$\overline{A}B\overline{C}D$	5	0	1	0	1	✓
$\overline{A}BC\overline{D}$	6	0	1	1	0	✓
$AB\overline{C}\,\overline{D}$	12	1	1	0	0	✓
$\overline{A}BCD$	7	0	1	1	1	✓
$A\overline{B}CD$	11	1	0	1	1	✓
$AB\overline{C}D$	13	1	1	0	1	✓
$ABCD$	15	1	1	1	1	✓

The new table is organized into groups, as indicated, in the same fashion as the first table. The second table is then processed in the same manner as the first. That is, terms that differ in only one variable are checked and a new term produced for a third table. In this example, the third table that is produced contains only one term: BD.

In general, the process would proceed through successive tables until a table with no matches was produced. In this case, this has involved three tables.

Once the process just described is completed, we have eliminated many of the possible terms of the expression. Those terms that have not been eliminated are used to construct a matrix, as illustrated in Table 11.6. Each row of the matrix corresponds to one of the terms that have not been eliminated (has no check) in any of the tables used so far. Each column corresponds to one of the terms in the original expression. An X is placed at each intersection of a row and a column such that the row element is "compatible" with the column element. That is, the variables present in the row element have the same value as the variables present in the column element. Next, circle each X that is alone in a column. Then place a square around each X in any row in which there is a circled X. If every column now has either a squared or a circled X, then we are done, and those row elements whose Xs have been marked constitute the minimal expression. Thus, in our example, the final expression is

$$AB\overline{C} + ACD + \overline{A}BC + \overline{A}\,\overline{C}D$$

In cases in which some columns have neither a circle nor a square, additional processing is required. Essentially, we keep adding row elements until all columns are covered.

Let us summarize the Quine–McCluskey method to try to justify intuitively why it works. The first phase of the operation is reasonably straightforward. The process eliminates unneeded variables in product terms. Thus, the expression $ABC + AB\overline{C}$ is equivalent to AB, because

$$ABC + AB\overline{C} = AB(C + \overline{C}) = AB$$

After the elimination of variables, we are left with an expression that is clearly equivalent to the original expression. However, there may be redundant terms in this expression, just as we found redundant groupings in Karnaugh maps. The matrix layout assures that each term in the original expression is covered and does so in a way that minimizes the number of terms in the final expression.

Table 11.6 Last Stage of Quine–McCluskey Method
(for $F = ABCD + AB\overline{C}D + AB\overline{C}\,\overline{D} + A\overline{B}CD + \overline{A}BCD + \overline{A}BC\overline{D} + \overline{A}B\overline{C}D + \overline{A}\,\overline{B}\,\overline{C}D$)

	ABCD	**AB$\overline{C}$D**	**AB$\overline{C}$ $\overline{D}$**	**A$\overline{B}$CD**	**$\overline{A}$BCD**	**$\overline{A}$BC$\overline{D}$**	**$\overline{A}$B$\overline{C}$D**	**$\overline{A}$ $\overline{B}$ $\overline{C}$D**
BD	X	X			X		X	
$\overline{A}\,\overline{C}D$							[X]	⊗
$\overline{A}BC$					[X]	⊗		
$AB\overline{C}$		[X]	⊗					
ACD	[X]			⊗				

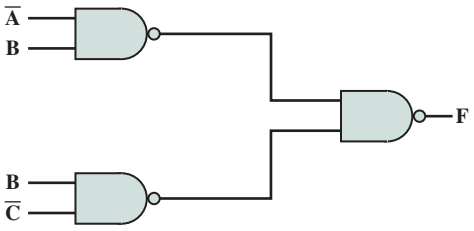


Figure 11.11 NAND Implementation of Table 11.3

NAND AND NOR IMPLEMENTATIONS Another consideration in the implementation of Boolean functions concerns the types of gates used. It is sometimes desirable to implement a Boolean function solely with NAND gates or solely with NOR gates. Although this may not be the minimum-gate implementation, it has the advantage of regularity, which can simplify the manufacturing process. Consider again Equation (11.3):

$$F = B(\overline{A} + \overline{C})$$

Because the complement of the complement of a value is just the original value,

$$F = B(\overline{A} + \overline{C}) = \overline{\overline{(\overline{A}B + (B\overline{C})}}$$

Applying DeMorgan's theorem,

$$F = \overline{\overline{(\overline{A}B)} \bullet \overline{(B\overline{C})}}$$

which has three NAND forms, as illustrated in Figure 11.11.

Multiplexers

The **multiplexer** connects multiple inputs to a single output. At any time, one of the inputs is selected to be passed to the output. A general block diagram representation is shown in Figure 11.12. This represents a 4-to-1 multiplexer. There are four input lines, labeled D0, D1, D2, and D3. One of these lines is selected to provide the output

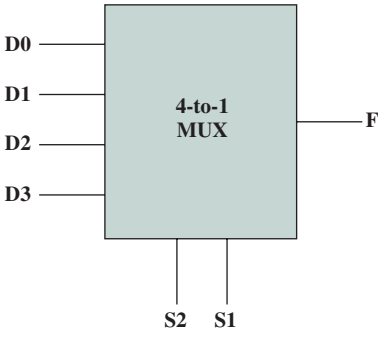


Figure 12.12 4-to-1 Multiplexer Representation

Table 11.7 4-to-1 Multiplexer Truth Table

S2	S1	F
0	0	D0
0	1	D1
1	0	D2
1	1	D3

signal F. To select one of the four possible inputs, a 2-bit selection code is needed, and this is implemented as two select lines labeled S1 and S2.

An example 4-to-1 multiplexer is defined by the truth table in Table 11.7. This is a simplified form of a truth table. Instead of showing all possible combinations of input variables, it shows the output as data from line D0, D1, D2, or D3. Figure 11.13 shows an implementation using AND, OR, and NOT gates. S1 and S2 are connected to the AND gates in such a way that, for any combination of S1 and S2, three of the AND gates will output 0. The fourth **AND gate** will output the value of the selected line, which is either 0 or 1. Thus, three of the inputs to the OR gate are always 0, and the output of the OR gate will equal the value of the selected input gate. Using this regular organization, it is easy to construct multiplexers of size 8-to-1, 16-to-1, and so on.

Multiplexers are used in digital circuits to control signal and data routing. An example is the loading of the program **counter** (PC). The value to be loaded into the program counter may come from one of several different sources:

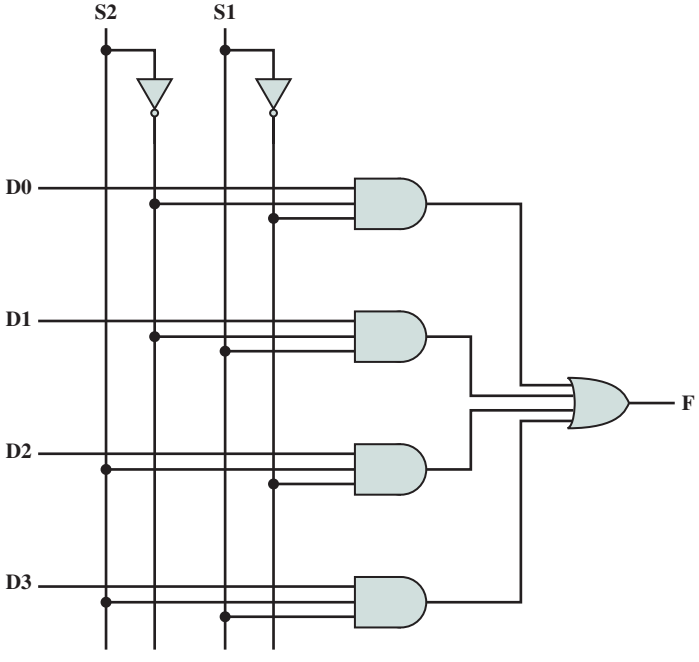


Figure 11.13 Multiplexer Implementation

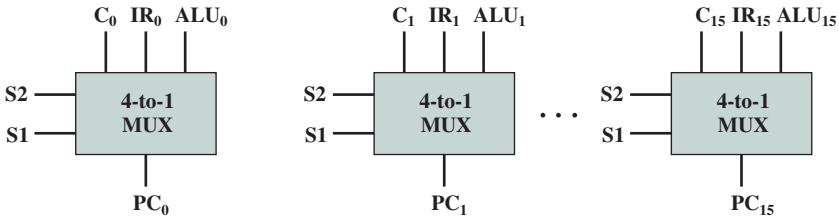


Figure 11.14 Multiplexer Input to Program Counter

- A binary counter, if the PC is to be incremented for the next instruction.
- The instruction **register**, if a branch instruction using a direct address has just been executed.
- The output of the ALU, if the branch instruction specifies the address using a displacement mode.

These various inputs could be connected to the input lines of a multiplexer, with the PC connected to the output line. The select lines determine which value is loaded into the PC. Because the PC contains multiple bits, multiple multiplexers are used, one per bit. Figure 11.14 illustrates this for 16-bit addresses.

Decoders

A **decoder** is a combinational circuit with a number of output lines, only one of which is asserted at any time. Which output line is asserted depends on the pattern of input lines. In general, a decoder has n inputs and 2^n outputs. Figure 11.15 shows a decoder with three inputs and eight outputs.

Decoders find many uses in digital computers. One example is address decoding. Suppose we wish to construct a 1K-byte memory using four 256×8-bit RAM chips. We want a single unified address space, which can be broken down as follows:

Address	Chip
0000–00FF	0
0100–01FF	1
0200–02FF	2
0300–03FF	3

Each chip requires 8 address lines, and these are supplied by the lower-order 8 bits of the address. The higher-order 2 bits of the 10-bit address are used to select one of the four RAM chips. For this purpose, a 2-to-4 decoder is used whose output enables one of the four chips, as shown in Figure 11.16.

With an additional input line, a decoder can be used as a demultiplexer. The demultiplexer performs the inverse function of a multiplexer; it connects a single input to one of several outputs. This is shown in Figure 11.17. As before, n inputs are decoded to produce a single one of 2^n outputs. All of the 2^n output lines are ANDed

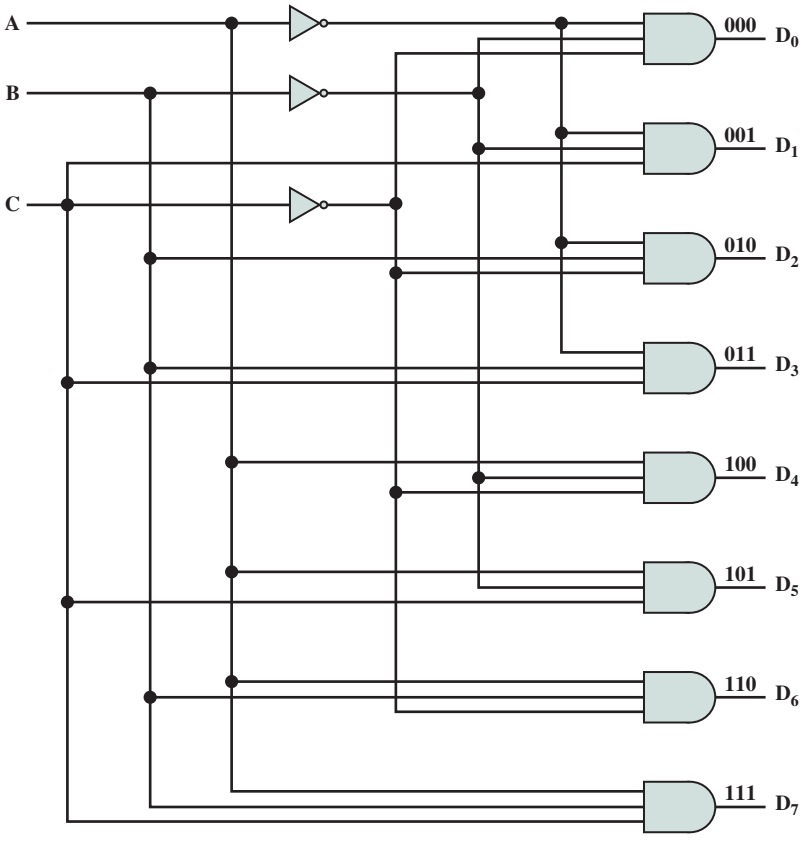


Figure 11.15 Decoder with 3 Inputs and $2^3 = 8$ Outputs

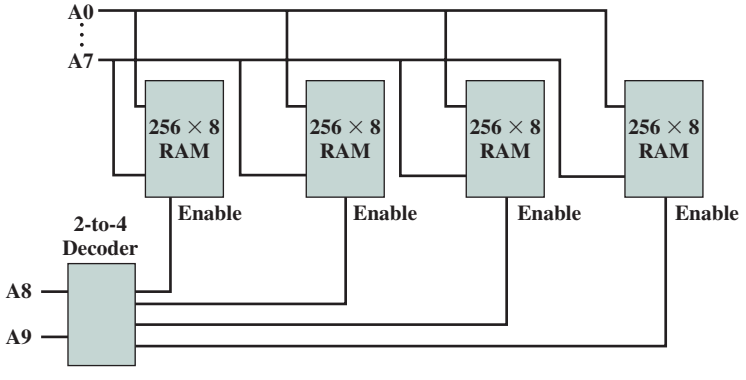


Figure 11.16 Address Decoding

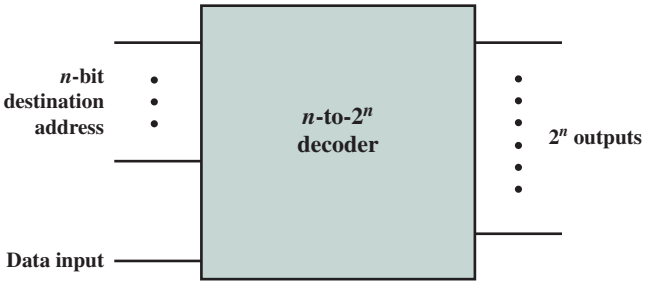


Figure 11.17 Implementation of a Demultiplexer Using a Decoder

with a data input line. Thus, the n inputs act as an address to select a particular output line, and the value on the data input line (0 or 1) is routed to that output line.

The configuration in Figure 11.17 can be viewed in another way. Change the label on the new line from *Data Input* to *Enable.* This allows for the control of the timing of the decoder. The decoded output appears only when the encoded input is present *and* the enable line has a value of 1.

Read-Only Memory

Combinational circuits are often referred to as "memoryless" circuits, because their output depends only on their current input and no history of prior inputs is retained. However, there is one sort of memory that is implemented with combinational circuits, namely **read-only memory (ROM)**.

Recall that a ROM is a memory unit that performs only the read operation. This implies that the binary information stored in a ROM is permanent and was created during the fabrication process. Thus, a given input to the ROM (address lines) always produces the same output (data lines). Because the outputs are a function only of the present inputs, the ROM is in fact a combinational circuit.

A ROM can be implemented with a decoder and a set of OR gates. As an example, consider Table 11.8. This can be viewed as a truth table with four inputs and four outputs. For each of the 16 possible input values, the corresponding set of values of the outputs is shown. It can also be viewed as defining the contents of a 64-bit ROM consisting of 16 words of 4 bits each. The four inputs specify an address, and the four outputs specify the contents of the location specified by the address. Figure 11.18 shows how this memory could be implemented using a 4-to-16 decoder and four OR gates. As with the PLA, a regular organization is used, and the interconnections are made to reflect the desired result.

Adders

So far, we have seen how interconnected gates can be used to implement such functions as the routing of signals, decoding, and ROM. One essential area not yet addressed is that of arithmetic. In this brief overview, we will content ourselves with looking at the addition function.

Binary addition differs from Boolean algebra in that the result includes a carry term. Thus,

Table 11.8 Truth Table for a ROM

Input				Output			
X_1	X_2	X_3	X_4	Z_1	Z_2	Z_3	Z_4
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

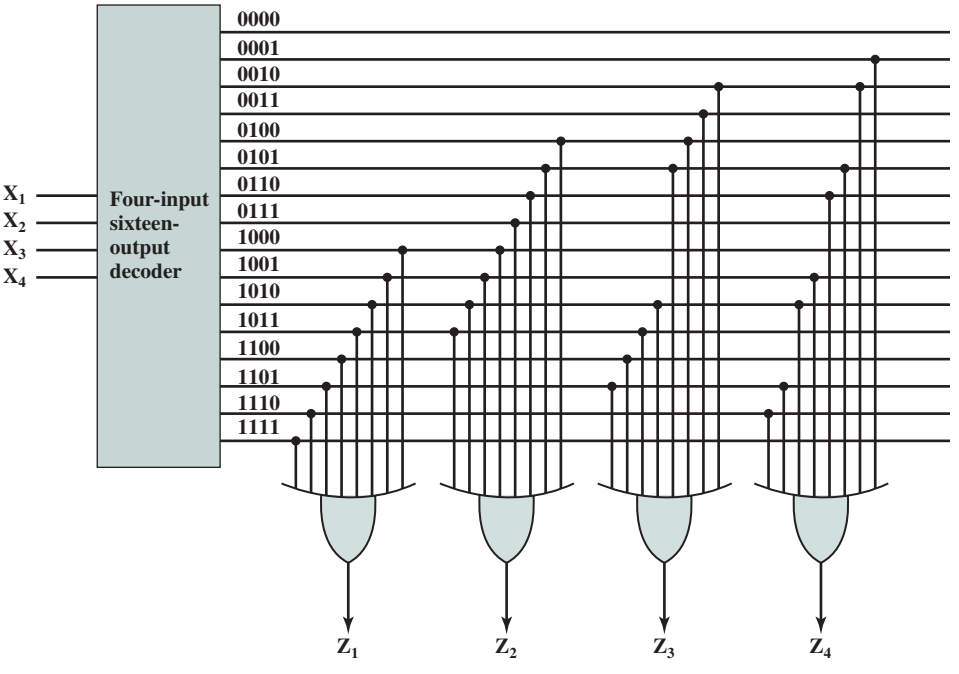


Figure 11.18 A 64-Bit ROM

Table 11.9 Binary Addition Truth Tables

(a) Single-Bit Addition

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(b) Addition with Carry Input

C_{in}	A	B	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

0	0	1	1
+0	+1	+0	+1
0	1	1	10

However, addition can still be dealt with in Boolean terms. In Table 11.9a, we show the logic for adding two input bits to produce a 1-bit sum and a carry bit. This truth table could easily be implemented in digital logic. However, we are not interested in performing addition on just a single pair of bits. Rather, we wish to add two n-bit numbers. This can be done by putting together a set of adders so that the carry from one **adder** is provided as input to the next. A 4-bit adder is depicted in Figure 11.19.

For a multiple-bit adder to work, each of the single-bit adders must have three inputs, including the carry from the next-lower-order adder. The revised truth table appears in Table 11.9b. The two outputs can be expressed:

$$\text{Sum} = \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + ABC + A\overline{B}\overline{C}$$

$$\text{Carry} = AB + AC + BC$$

Figure 11.20 is an implementation using AND, OR, and NOT gates.

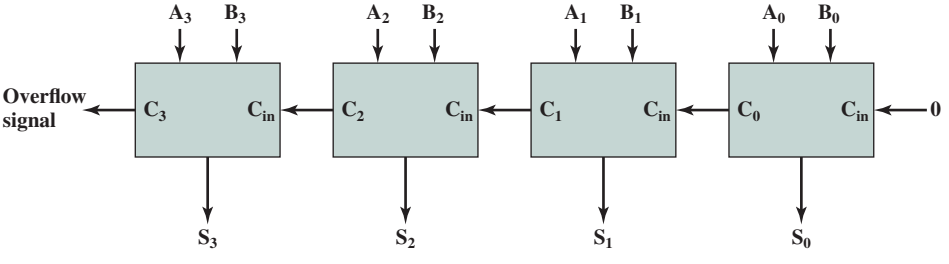


Figure 11.19 4-Bit Adder

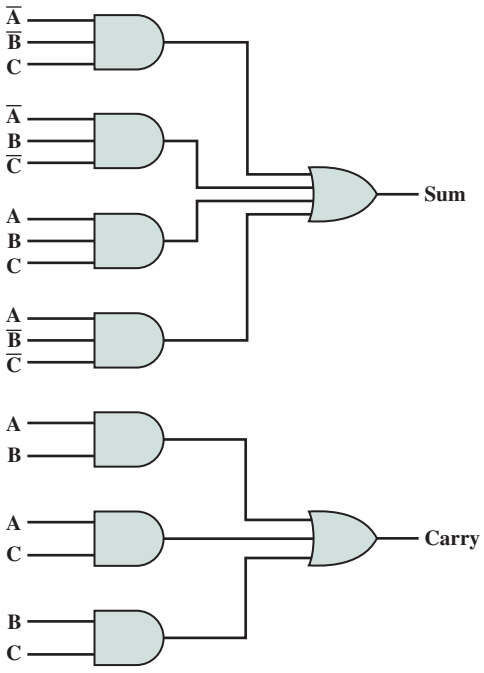


Figure 11.20 Implementation of an Adder

Thus we have the necessary logic to implement a multiple-bit adder such as shown in Figure 11.21. Note that because the output from each adder depends on the carry from the previous adder, there is an increasing delay from the least significant to the most significant bit. Each single-bit adder experiences a certain amount of gate delay, and this gate delay accumulates. For larger adders, the accumulated delay can become unacceptably high.

If the carry values could be determined without having to ripple through all the previous stages, then each single-bit adder could function independently, and delay would not accumulate. This can be achieved with an approach known as *carry lookahead.* Let us look again at the 4-bit adder to explain this approach.

We would like to come up with an expression that specifies the carry input to any stage of the adder without reference to previous carry values. We have

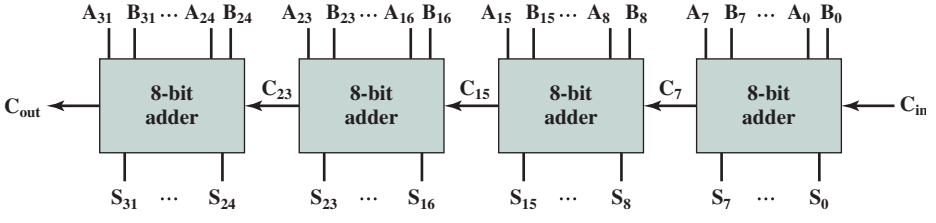


Figure 11.21 Construction of a 32-Bit Adder Using 8-Bit Adders

$$C_0 = A_0B_0 \tag{11.4}$$

$$C_1 = A_1B_1 + A_1A_0B_0 + B_1A_0B_0 \tag{11.5}$$

Following the same procedure, we get

$$C_2 = A_2B_2 + A_2A_1B_1 + A_2A_1A_0B_0 + A_2B_1A_0B_0 + B_2A_1B_1 + B_2A_1A_0B_0 + B_2B_1A_0B_0$$

This process can be repeated for arbitrarily long adders. Each carry term can be expressed in SOP form as a function only of the original inputs, with no dependence on the carries. Thus, only two levels of gate delay occur regardless of the length of the adder.

For long numbers, this approach becomes excessively complicated. Evaluating the expression for the most significant bit of an n-bit adder requires an OR gate with $2^n - 1$ inputs and $2^n - 1$ AND gates with from 2 to $n + 1$ inputs. Accordingly, full carry lookahead is typically done only 4 to 8 bits at a time. Figure 11.21 shows how a 32-bit adder can be constructed out of four 8-bit adders. In this case, the carry must ripple through the four 8-bit adders, but this will be substantially quicker than a ripple through thirty-two 1-bit adders.

11.4 SEQUENTIAL CIRCUITS

Combinational circuits implement the essential functions of a digital computer. However, except for the special case of ROM, they provide no memory or state information, elements also essential to the operation of a digital computer. For the latter purposes, a more complex form of digital logic circuit is used: the **sequential circuit**. The current output of a sequential circuit depends not only on the current input, but also on the past history of inputs. Another and generally more useful way to view it is that the current output of a sequential circuit depends on the current input and the current state of that circuit.

In this section, we examine some simple but useful examples of sequential circuits. As will be seen, the sequential circuit makes use of combinational circuits.

Flip-Flops

The simplest form of sequential circuit is the **flip-flop**. There are a variety of flip-flops, all of which share two properties:

- The flip-flop is a bistable device. It exists in one of two states and, in the absence of input, remains in that state. Thus, the flip-flop can function as a 1-bit memory.
- The flip-flop has two outputs, which are always the complements of each other. These are generally labeled Q and $\overline{Q}$.

THE S–R LATCH Figure 11.22 shows a common configuration known as the S–R flip-flop or **S–R latch**. The circuit has two inputs, S (Set) and R (Reset), and two outputs, Q and $\overline{Q}$, and consists of two NOR gates connected in a feedback arrangement.

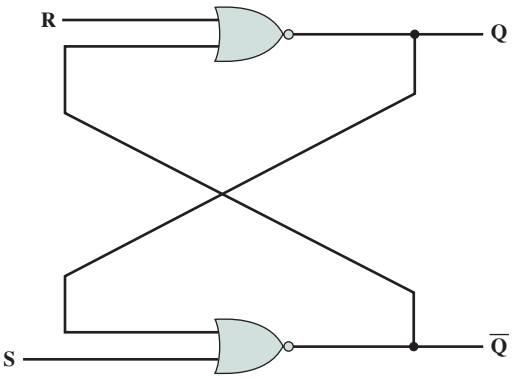


Figure 11.22 The S–R Latch Implemented with NOR Gates

First, let us show that the circuit is bistable. Assume that both S and R are 0 and that Q is 0. The inputs to the lower NOR gate are $Q = 0$ and $S = 0$. Thus, the output $\overline{Q} = 1$ means that the inputs to the upper NOR gate are $\overline{Q} = 1$ and $R = 0$, which has the output $Q = 0$. Thus, the state of the circuit is internally consistent and remains stable as long as $S = R = 0$. A similar line of reasoning shows that the state $Q = 1, \overline{Q} = 0$ is also stable for $R = S = 0$.

Thus, this circuit can function as a 1-bit memory. We can view the output Q as the "value" of the bit. The inputs S and R serve to write the values 1 and 0, respectively, into memory. To see this, consider the state $Q = 0, \overline{Q} = 1, S = 0, R = 0$. Suppose that S changes to the value 1. Now the inputs to the lower NOR gate are $S = 1, Q = 0$. After some time delay Δt, the output of the lower NOR gate will be $\overline{Q} = 0$ (see Figure 11.23). So, at this point in time, the inputs to the upper NOR gate become $R = 0, \overline{Q} = 0$. After another gate delay of Δt the output Q becomes 1. This is again a stable state. The inputs to the lower gate are now $S = 1, Q = 1$, which maintain the output $\overline{Q} = 0$. As long as $S = 1$ and $R = 0$, the outputs will remain $Q = 1, \overline{Q} = 0$. Furthermore, if S returns to 0, the outputs will remain unchanged.

The R output performs the opposite function. When R goes to 1, it forces $Q = 0, \overline{Q} = 1$ regardless of the previous state of Q and $\overline{Q}$. Again, a time delay of $2\Delta t$ occurs before the final state is established (Figure 11.23).

The S–R latch can be defined with a table similar to a truth table, called a *characteristic table*, which shows the next state or states of a sequential circuit as a function of current states and inputs. In the case of the S–R latch, the state can be defined by the value of Q. Table 11.10a shows the resulting characteristic table. Observe that the inputs $S = 1, R = 1$ are not allowed, because these would produce an inconsistent output (both Q and $\overline{Q}$ equal 0). The table can be expressed more compactly, as in Table 11.10b. An illustration of the behavior of the S–R latch is shown in Table 11.10c.

CLOCKED S–R FLIP–FLOP The output of the S–R latch changes, after a brief time delay, in response to a change in the input. This is referred to as asynchronous operation. More typically, events in the digital computer are synchronized to a clock pulse, so that changes occur only when a clock pulse occurs. Figure 11.24 shows this

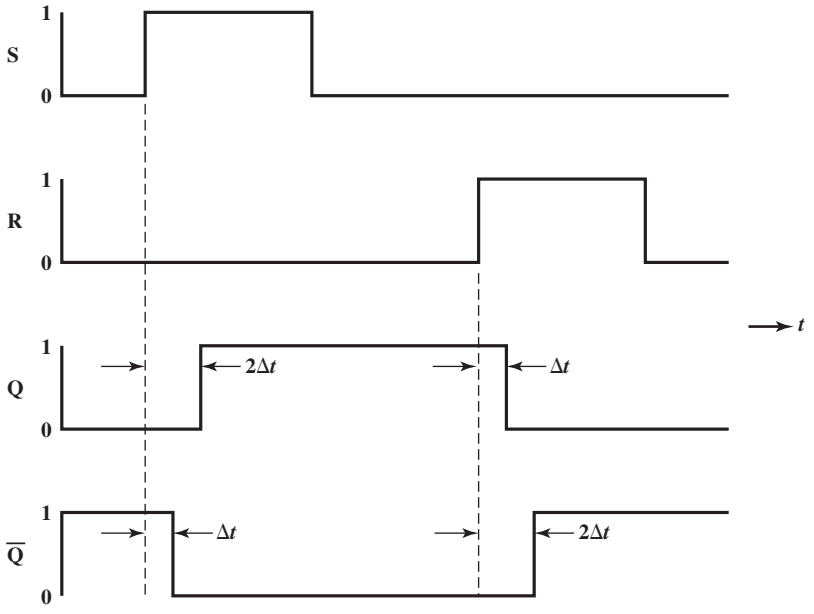


Figure 11.23 NOR S–R Latch Timing Diagram

Table 11.10 The S–R Latch

(a) Characteristic Table		
Current Inputs	**Current State**	**Next State**
SR	Q_n	Q_{n+1}
00	0	0
00	1	1
01	0	0
01	1	0
10	0	1
10	1	1
11	0	—
11	1	—

(b) Simplified Characteristic Table		
S	**R**	$\mathbf{Q_{n+1}}$
0	0	Q_n
0	1	0
1	0	1
1	1	—

(c) Response to Series of Inputs										
t	0	1	2	3	4	5	6	7	8	9
S	1	0	0	0	0	0	0	0	1	0
R	0	0	0	1	0	0	1	0	0	0
$\mathbf{Q_{n+1}}$	1	1	1	0	0	0	0	0	1	1

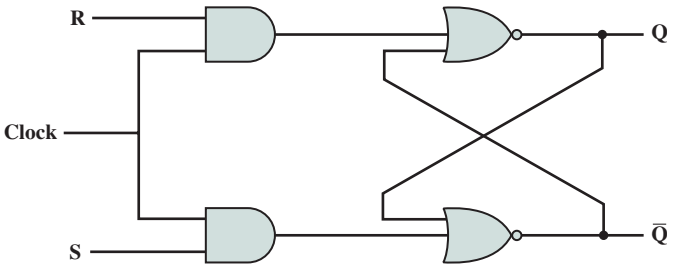


Figure 11.24 Clocked S–R Flip-Flop

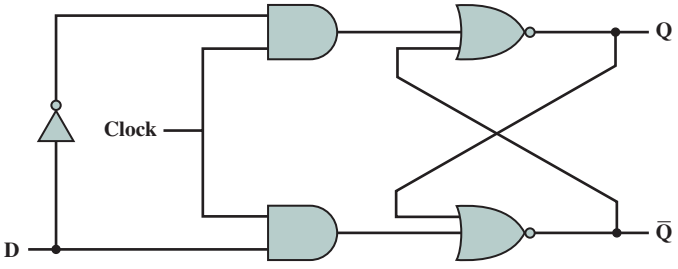


Figure 11.25 D Flip-Flop

arrangement. This device is referred to as a **clocked S–R flip-flop**. Note that the R and S inputs are passed to the NOR gates only during the clock pulse.

D FLIP–FLOP One problem with S–R flip-flop is that the condition R = 1, S = 1 must be avoided. One way to do this is to allow just a single input. The **D flip-flop** accomplishes this. Figure 11.25 shows a gate implementation of the D flip-flop. By using an inverter, the nonclock inputs to the two AND gates are guaranteed to be the opposite of each other.

The D flip-flop is sometimes referred to as the data flip-flop because it is, in effect, storage for one bit of data. The output of the D flip-flop is always equal to the most recent value applied to the input. Hence, it remembers and produces the last input. It is also referred to as the delay flip-flop, because it delays a 0 or 1 applied to its input for a single clock pulse. We can capture the logic of the D flip-flop in the following truth table:

D	Q_{n+1}
0	0
1	1

J–K FLIP–FLOP Another useful flip-flop is the **J–K flip-flop**. Like the S–R flip-flop, it has two inputs. However, in this case all possible combinations of input values are valid. Figure 11.26 shows a gate implementation of the J–K flip-flop, and Figure 11.27 shows its characteristic table (along with those for the S–R and D flip-flops). Note that the first three combinations are the same as for the S–R flip-flop. With no input asserted, the output is stable. If only the J input is asserted, the result is a set

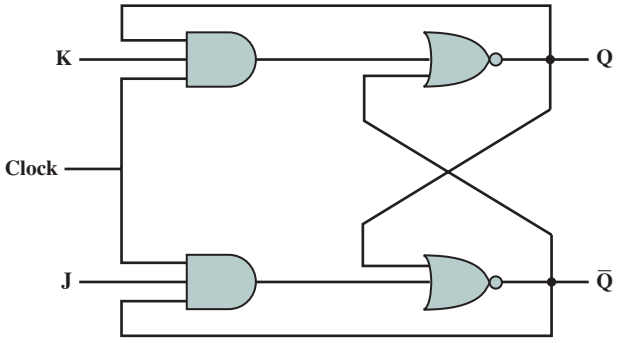


Figure 11.26 J–K Flip-Flop

function, causing the output to be 1; if only the K input is asserted, the result is a reset function, causing the output to be 0. When both J and K are 1, the function performed is referred to as the toggle function: the output is reversed. Thus, if Q is 1 and 1 is applied to J and K, then Q becomes 0. The reader should verify that the implementation of Figure 11.26 produces this characteristic function.

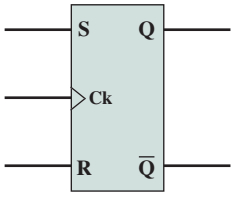
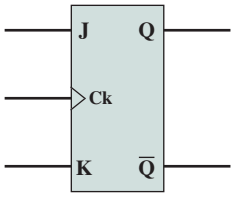
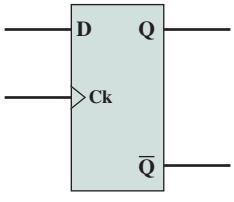
Name	Graphical Symbol	Truth Table
S–R		S R Q_{n+1} 0 0 Q_n 0 1 0 1 0 1 1 1 –
J–K		J K Q_{n+1} 0 0 Q_n 0 1 0 1 0 1 1 1 $\overline{Q_n}$
D		D Q_{n+1} 0 0 1 1

Figure 11.27 Basic Flip-Flops

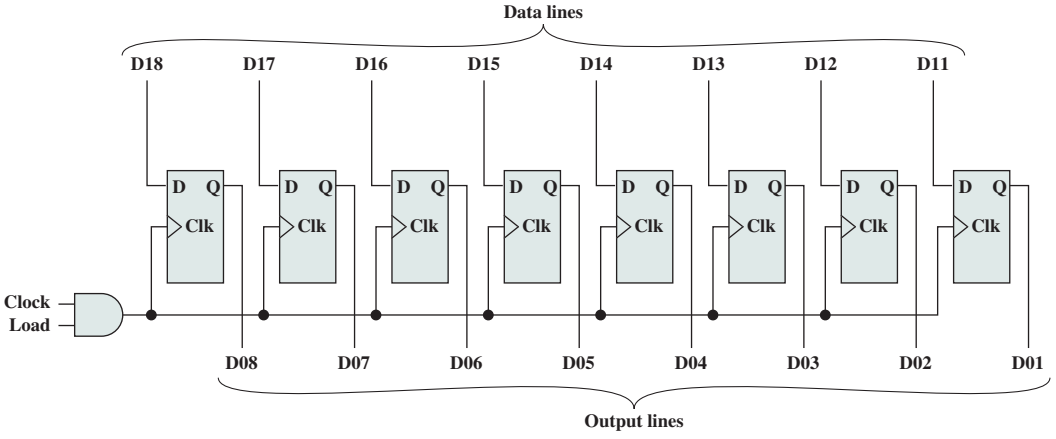


Figure 11.28 8-Bit Parallel Register

Registers

As an example of the use of flip-flops, let us first examine one of the essential elements of the CPU: the register. As we know, a register is a digital circuit used within the CPU to store one or more bits of data. Two basic types of registers are commonly used: parallel registers and shift registers.

PARALLEL REGISTERS A **parallel register** consists of a set of 1-bit memories that can be read or written simultaneously. It is used to store data. The registers that we have discussed throughout this book are parallel registers.

The 8-bit register of Figure 11.28 illustrates the operation of a parallel register using D flip-flops. A control signal, labeled *load*, controls writing into the register from signal lines, D11 through D18. These lines might be the output of multiplexers, so that data from a variety of sources can be loaded into the register.

SHIFT REGISTER A **shift register** accepts and/or transfers information serially. Consider, for example, Figure 11.29, which shows a 5-bit shift register constructed from clocked D flip-flops. Data are input only to the leftmost flip-flop. With each clock pulse, data are shifted to the right one position, and the rightmost bit is transferred out.

Shift registers can be used to interface to serial I/O devices. In addition, they can be used within the ALU to perform logical shift and rotate functions. In this

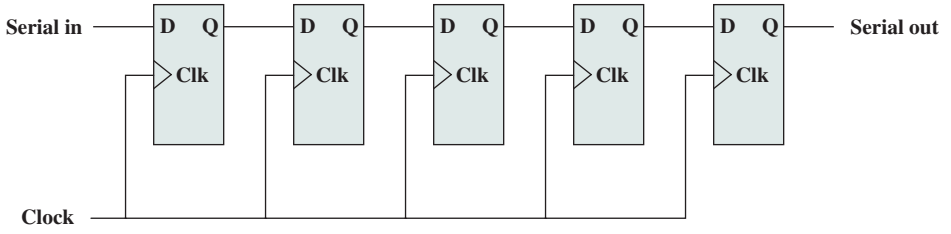


Figure 11.29 5-Bit Shift Register

latter capacity, they need to be equipped with parallel read/write circuitry as well as serial.

Counters

Another useful category of sequential circuit is the **counter**. A counter is a register whose value is easily incremented by 1 modulo the capacity of the register; that is, after the maximum value is achieved the next increment sets the counter value to 0. Thus, a register made up of n flip-flops can count up to $2^n - 1$. An example of a counter in the CPU is the program counter.

Counters can be designated as asynchronous or synchronous, depending on the way in which they operate. Asynchronous counters are relatively slow because the output of one flip-flop triggers a change in the status of the next flip-flop. In a **synchronous counter**, all of the flip-flops change state at the same time. Because the latter type is much faster, it is the kind used in CPUs. However, it is useful to begin the discussion with a description of an asynchronous counter.

RIPPLE COUNTER An asynchronous counter is also referred to as a **ripple counter**, because the change that occurs to increment the counter starts at one end and "ripples" through to the other end. Figure 11.30 shows an implementation of a 4-bit counter using J–K flip-flops, together with a timing diagram that illustrates its behavior. The timing diagram is idealized in that it does not show the propagation delay that occurs as the signals move down the series of flip-flops. The output of the leftmost flip-flop (Q_0) is the least significant bit. The design could clearly be extended to an arbitrary number of bits by cascading more flip-flops.

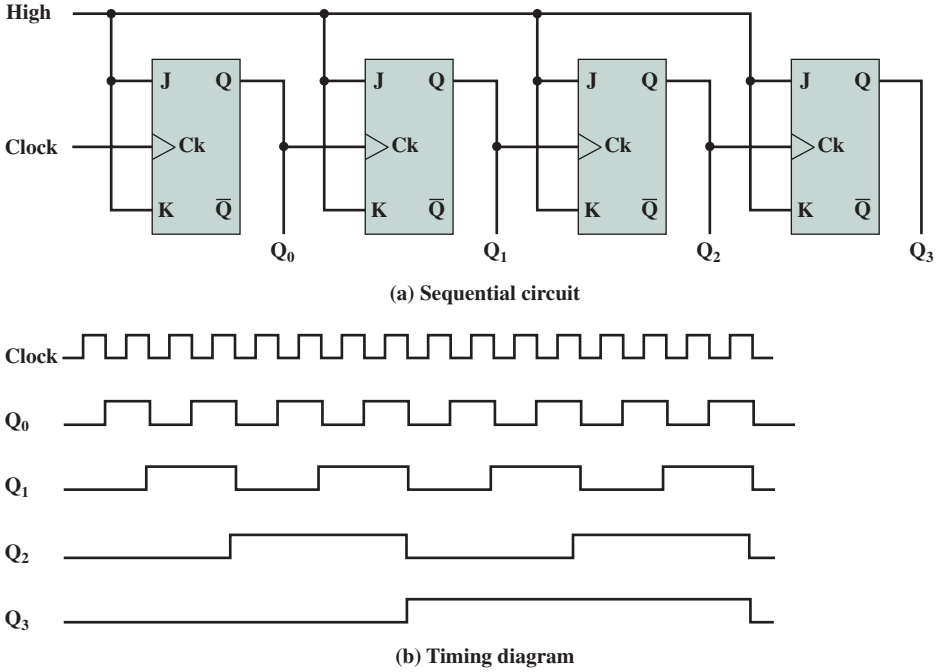


Figure 11.30 Ripple Counter

In the illustrated implementation, the counter is incremented with each clock pulse. The J and K inputs to each flip-flop are held at a constant 1. This means that, when there is a clock pulse, the output at Q will be inverted (1 to 0; 0 to 1). Note that the change in state is shown as occurring with the falling edge of the clock pulse; this is known as an edge-triggered flip-flop. Using flip-flops that respond to the transition in a clock pulse rather than the pulse itself provides better timing control in complex circuits. If one looks at patterns of output for this counter, it can be seen that it cycles through 0000, 0001, …, 1110, 1111, 0000, and so on.

SYNCHRONOUS COUNTERS The ripple counter has the disadvantage of the delay involved in changing value, which is proportional to the length of the counter. To overcome this disadvantage, CPUs make use of synchronous counters, in which all of the flip-flops of the counter change at the same time. In this subsection, we present a design for a 3-bit synchronous counter. In doing so, we illustrate some basic concepts in the design of a synchronous circuit.

For a 3-bit counter, three flip-flops will be needed. Let us use J–K flip-flops. Label the uncomplemented output of the three flip-flops C, B, and A, respectively, with C representing the most significant bit. The first step is to construct a truth table that relates the J–K inputs and outputs, to allow us to design the overall circuit. Such a truth table is shown in Figure 11.31a. The first three columns show the possible combinations of outputs C, B, and A. They are listed in the order that they will appear as the counter is incremented. Each row lists the current value of C, B, and A and the inputs to the three flip-flops that will be required to reach the next value of C, B, and A.

To understand the way in which the truth table of Figure 11.31a is constructed, it may be helpful to recast the characteristic table for the J–K flip-flop. Recall that this table was presented as follows:

J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	$\overline{Q_{n+1}}$

In this form, the table shows the effect that the J and K inputs have on the output. Now consider the following organization of the same information:

Q_n	J	K	Q_{n+1}
0	0	d	0
0	1	d	1
1	d	1	0
1	d	0	1

In this form, the table provides the value of the next output when the inputs and the present output are known. This is exactly the information needed to design the counter or, indeed, any sequential circuit. In this form, the table is referred to as an **excitation table**.

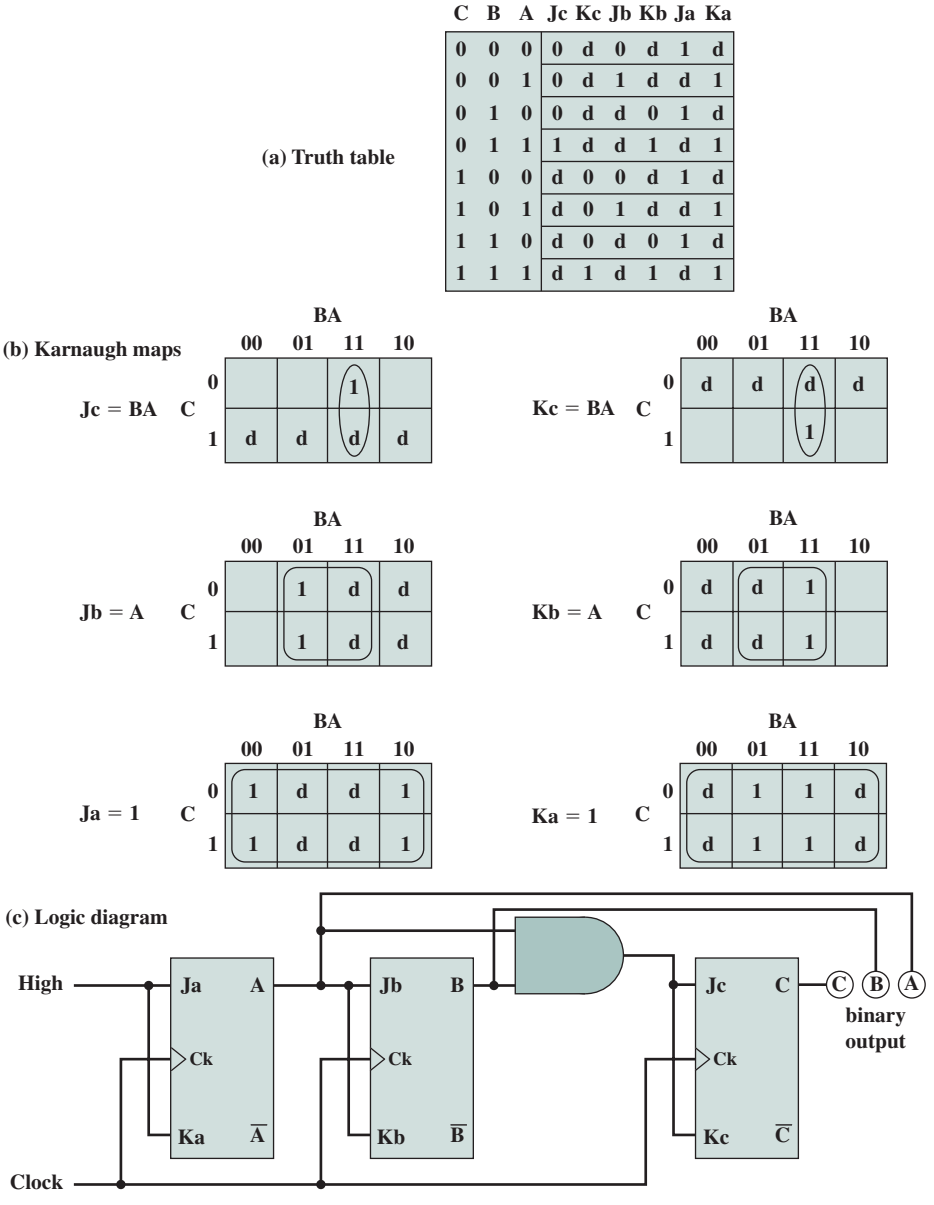

(a) Truth table
C B A Jc Kc Jb Kb Ja Ka
0 0 0 0 d 0 d 1 d
0 0 1 0 d 1 d d 1
0 1 0 0 d d 0 1 d
0 1 1 1 d d 1 d 1
1 0 0 d 0 0 d 1 d
1 0 1 d 0 1 d d 1
1 1 0 d 0 d 0 1 d
1 1 1 d 1 d 1 d 1
(b) Karnaugh maps
Jc = BA C
Kc = BA C
Jb = A C
Kb = A C
Ja = 1 C
Ka = 1 C
BA
00 01 11 10
(c) Logic diagram
High
Clock
Ja A
Ck
Ka $\overline{A}$
Jb B
Kb $\overline{B}$
Jc C
Kc $\overline{C}$
C B A
binary output

Figure 11.31 Design of a Synchronous Counter

Let us return to Figure 11.31a. Consider the first row. We want the value of C to remain 0, the value of B to remain 0, and the value of A to go from 0 to 1 with the next application of a clock pulse. The excitation table shows that to maintain an output of 0, we must have inputs of J = 0 and don't care for K. To effect a transition from 0 to 1, the inputs must be J = 1 and K = d. These values are shown in the first row of the table. By similar reasoning, the remainder of the table can be filled in.

Having constructed the truth table of Figure 11.31a, we see that the table shows the required values of all of the J and K inputs as functions of the current values of C, B, and A. With the aid of Karnaugh maps, we can develop Boolean expressions for these six functions. This is shown in part b of the figure. For example, the Karnaugh map for the variable Ja (the J input to the flip-flop that produces the A output) yields the expression Ja = BC. When all six expressions are derived, it is a straightforward matter to design the actual circuit, as shown in part c of the figure.

11.5 PROGRAMMABLE LOGIC DEVICES

Thus far, we have treated individual gates as building blocks, from which arbitrary functions can be realized. The designer could pursue a strategy of minimizing the number of gates to be used by manipulating the corresponding Boolean expressions.

As the level of integration provided by integrated circuits increases, other considerations apply. Early integrated circuits, using small-scale integration (SSI), provided from one to ten gates on a chip. Each gate is treated independently, in the building-block approach described so far. To construct a logic function, a number of these chips are laid out on a printed circuit board and the appropriate pin interconnections are made.

Increasing levels of integration made it possible to put more gates on a chip and to make gate interconnections on the chip as well. This yields the advantages of decreased cost, decreased size, and increased speed (because on-chip delays are of shorter duration than off-chip delays). A design problem arises, however. For each particular logic function or set of functions, the layout of gates and interconnections on the chip must be designed. The cost and time involved in such custom chip design is high. Thus, it becomes attractive to develop a general-purpose chip that can be readily adapted to specific purposes. This is the intent of the *programmable logic device* (PLD).

There are a number of different types of PLDs in commercial use. Table 11.11 lists some of the key terms and defines some of the most important types. In this section, we first look at one of the simplest such devices, the *programmable logic array (PLA)* and then introduce perhaps the most important and widely used type of PLD, the field-programmable gate array (FPGA).

Programmable Logic Array

The PLA is based on the fact that any Boolean function (truth table) can be expressed in a sum-of-products (SOP) form, as we have seen. The PLA consists of a regular arrangement of NOT, AND, and OR gates on a chip. Each chip input is passed through a NOT gate so that each input and its complement are available to each AND gate. The output of each AND gate is available to each OR gate, and the output of each OR gate is a chip output. By making the appropriate connections, arbitrary SOP expressions can be implemented.

Figure 11.32a shows a PLA with three inputs, eight gates, and two outputs. On the left is a programmable AND array. The AND array is programmed by establishing a connection between any PLA input or its negation and any AND gate input by connecting the corresponding lines at their point of intersection. On the

Table 11.11 PLD Terminology

Programmable Logic Device (PLD) A general term that refers to any type of integrated circuit used for implementing digital hardware, where the chip can be configured by the end user to realize different designs. Programming of such a device often involves placing the chip into a special programming unit, but some chips can also be configured "in-system." Also referred to as a field-programmable device (FPD).
Programmable Logic Array (PLA) A relatively small PLD that contains two levels of logic, an AND-plane and an OR-plane, where both levels are programmable.
Programmable Array Logic (PAL) A relatively small PLD that has a programmable AND-plane followed by a fixed OR-plane.
Simple PLD (SPLD) A PLA or PAL.
Complex PLD (CPLD) A more complex PLD that consists of an arrangement of multiple SPLD-like blocks on a single chip.
Field-Programmable Gate Array (FPGA) A PLD featuring a general structure that allows very high logic capacity. Whereas CPLDs feature logic resources with a wide number of inputs (AND planes), FPGAs offer more narrow logic resources. FPGAs also offer a higher ratio of flip-flops to logic resources than do CPLDs.
Logic Block A relatively small circuit block that is replicated in an array in an FPD. When a circuit is implemented in an FPD, it is first decomposed into smaller subcircuits that can each be mapped into a logic block. The term *logic block* is mostly used in the context of FPGAs, but it could also refer to a block of circuitry in a CPLD.

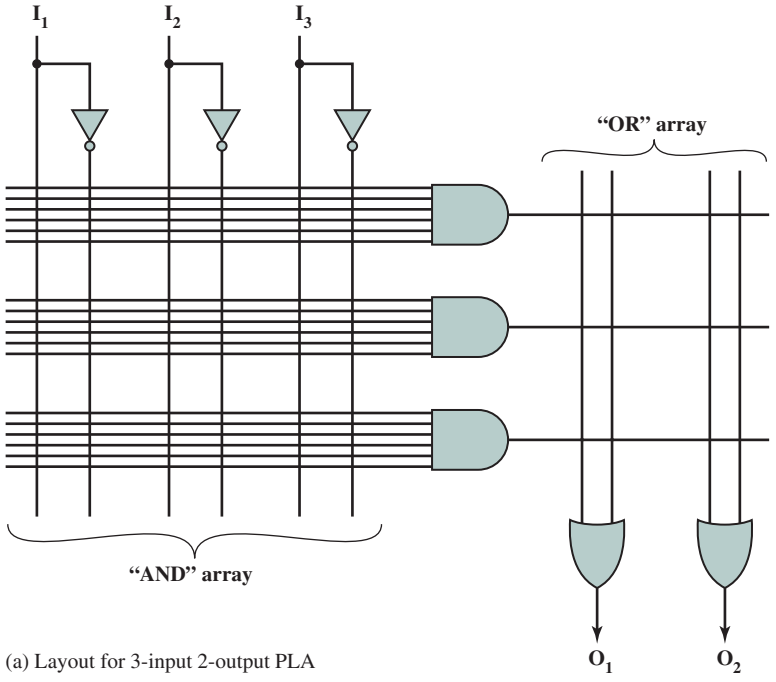
right is a programmable OR array, which involves connecting AND gate outputs to OR gate inputs. Most larger PLAs contain several hundred gates, 15 to 25 inputs, and 5 to 15 outputs. The connections from the inputs to the AND gates, and from the AND gates to the OR gates, are not specified until programming time.

PLAs are manufactured in two different ways to allow easy programming (making of connections). In the first, every possible connection is made through a fuse at every intersection point. The undesired connections can then be later removed by blowing the fuses. This type of PLA is referred to as a *field-programmable logic array (FPLA).* Alternatively, the proper connections can be made during chip fabrication by using an appropriate mask supplied for a particular interconnection pattern. In either case, the PLA provides a flexible, inexpensive way of implementing digital logic functions.

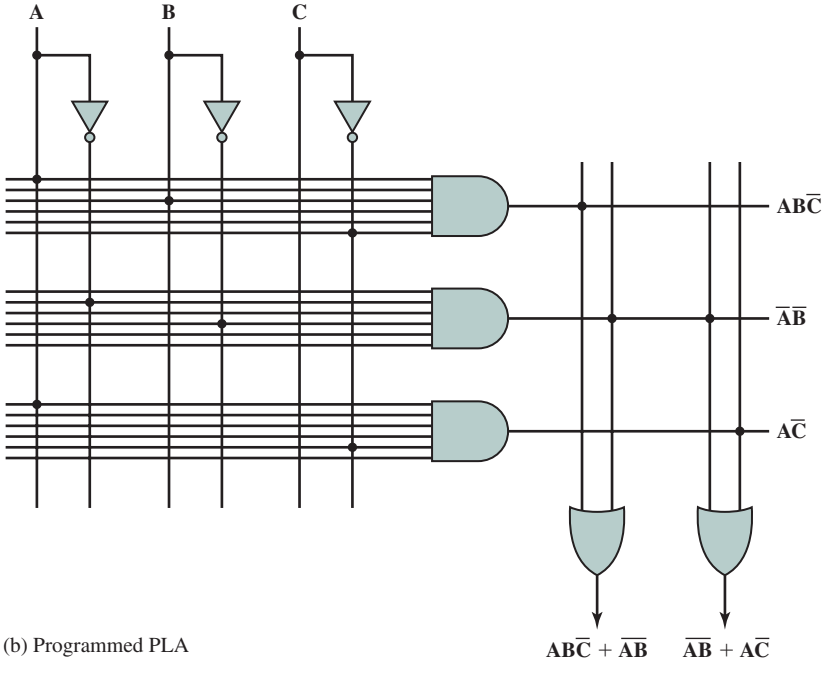
Figure 11.32b shows a programmed PLA that realizes two Boolean expressions.

Field-Programmable Gate Array

The PLA is an example of a simple PLD (SPLD). The difficulty with increasing capacity of a strict SPLD architecture is that the structure of the programmable logic-planes grows too quickly in size as the number of inputs is increased. The only feasible way to provide large capacity devices based on SPLD architectures is to then integrate multiple SPLDs onto a single chip and provide interconnect to programmably connect the SPLD blocks together. Many commercial PLD products



(a) Layout for 3-input 2-output PLA



(b) Programmed PLA

Figure 11.32 An Example of a Programmable Logic Array (PLA)

exist on the market today with this basic structure, and are collectively referred to as Complex PLDs (CPLDs). The most important type of CPLD is the FPGA.

An FPGA consists of an array of uncommitted circuit elements, called **logic blocks**, and interconnect resources. An illustration of a typical FPGA architecture is shown in Figure 11.33. The key components of an FPGA are;

- **Logic block:** The configurable logic blocks are where the computation of the user's circuit takes place.
- **I/O block:** The I/O blocks connect I/O pins to the circuitry on the chip.
- **Interconnect:** These are signal paths available for establishing connections among I/O blocks and logic blocks.

The logic block can be either a combinational circuit or a sequential circuit. In essence, the programming of a logic block is done by downloading the contents of a truth table for a logic function. Figure 11.34 shows an example of a simple logic block consisting of a D flip-flop, a 2-to-1 multiplexer, and a 16-bit **lookup table**. The lookup table is a memory consisting of 16 1-bit elements, so that 4 input lines are required to select one of the 16 bits. Larger logic blocks have larger lookup tables and multiple interconnected lookup tables. The combinational logic realized by the lookup table can be output directly or stored in the D flip-flop and output synchronously. A separate one-bit memory controls the multiplexer to determine whether the output comes directly from the lookup table or from the flip-flop.

By interconnecting numerous logic blocks, very complex logic functions can be easily implemented.

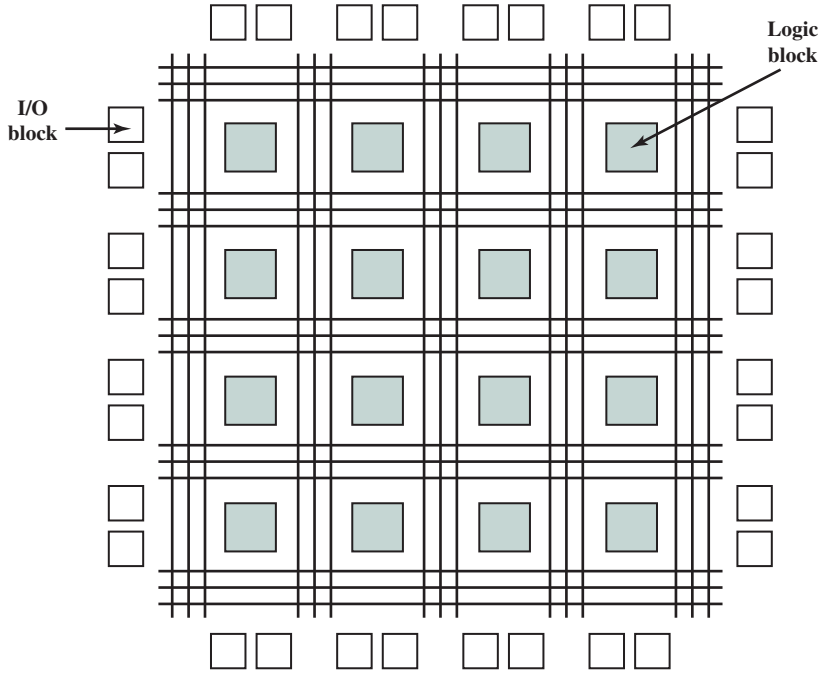


Figure 11.33 Structure of an FPGA

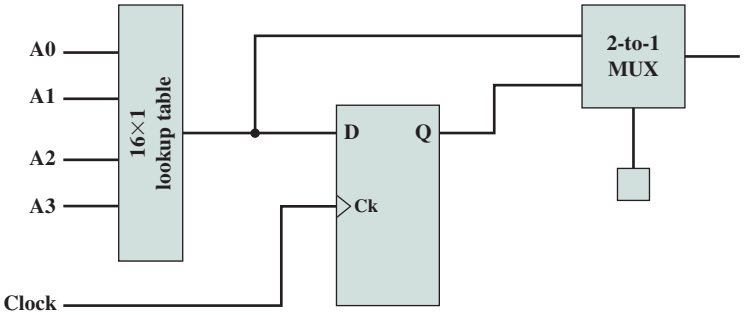


Figure 11.34 A Simple FPGA Logic Block

11.6 KEY TERMS AND PROBLEMS

Key Terms

adder	OR gate	register
AND gate	parallel register	excitation table
assert	combinational circuit	field-programmable gate array (FPGA)
Boolean algebra	complex PLD (CPLD)	flip-flop
clocked S–R flip-flop	counter	ripple counter
D flip-flop	decoder	sequential circuit
gates	product of sums (POS)	shift register
graphical symbol	programmable array logic (PAL)	simple PLD (SPLD)
J–K flip-flop	programmable logic array (PLA)	sum of products (SOP)
Karnaugh map	programmable logic device (PLD)	synchronous counter
logic block	Quine–McCluskey method	S–R Latch
lookup table	read-only memory (ROM)	truth table
multiplexer		XOR gate
NAND gate		
NOR		

Problems

11.1 Construct a truth table for the following Boolean expressions:

a. $ABC + \overline{A}\,\overline{B}\,\overline{C}$

b. $ABC + A\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,\overline{C}$

c. $A(B\overline{C} + \overline{B}C)$

d. $(A + B)(A + C)(\overline{A} + \overline{B})$

11.2 Simplify the following expressions according to the commutative law:

a. $A \cdot \overline{B} + \overline{B} \cdot A + C \cdot D \cdot E + \overline{C} \cdot D \cdot E + E \cdot \overline{C} \cdot D$

b. $A \cdot B + A \cdot C + B \cdot A$

c. $(L \cdot M \cdot N)(A \cdot B)(C \cdot D \cdot \overline{E})(M \cdot N \cdot L)$

d. $F \cdot (K + R) + S \cdot V + W \cdot \overline{X} + V \cdot S + \overline{X} \cdot W + (R + K) \cdot F$

11.3 Apply DeMorgan's theorem to the following equations:
- **a.** $F = \overline{V + A + L}$
- **b.** $F = \overline{A} + \overline{B} + \overline{C} + \overline{D}$

11.4 Simplify the following expressions:
- **a.** $A = S \cdot T + V \cdot W + R \cdot S \cdot T$
- **b.** $A = T \cdot U \cdot V + X \cdot Y + Y$
- **c.** $A = F \cdot (E + F + G)$
- **d.** $A = (P \cdot Q + R + S \cdot T)T \cdot S$
- **e.** $A = \overline{\overline{D} \cdot \overline{D} \cdot E}$
- **f.** $A = Y \cdot (W + X + \overline{\overline{Y} + \overline{Z}}) \cdot Z$
- **g.** $A = (B \cdot E + C + F) \cdot C$

11.5 Construct the operation XOR from the basic Boolean operations AND, OR, and NOT.

11.6 Given a NOR gate and NOT gates, draw a logic diagram that will perform the three-input AND function.

11.7 Write the Boolean expression for a four-input **NAND gate**.

11.8 A combinational circuit is used to control a seven-segment display of decimal digits, as shown in Figure 11.35. The circuit has four inputs, which provide the four-bit code used in packed decimal representation ($0_{10} = 0000, \ldots, 9_{10} = 1001$). The seven outputs define which segments will be activated to display a given decimal digit. Note that some combinations of inputs and outputs are not needed.
- **a.** Develop a truth table for this circuit.
- **b.** Express the truth table in SOP form.
- **c.** Express the truth table in POS form.
- **d.** Provide a simplified expression.

11.9 Design an 8-to-1 multiplexer.

11.10 Add an additional line to Figure 11.15 so that it functions as a demultiplexer.

11.11 The Gray code is a binary code for integers. It differs from the ordinary binary representation in that there is just a single bit change between the representations of any two numbers. This is useful for applications such as counters or analog-to-digital converters where a sequence of numbers is generated. Because only one bit changes at a time, there is never any ambiguity due to slight timing differences. The first eight elements of the code are

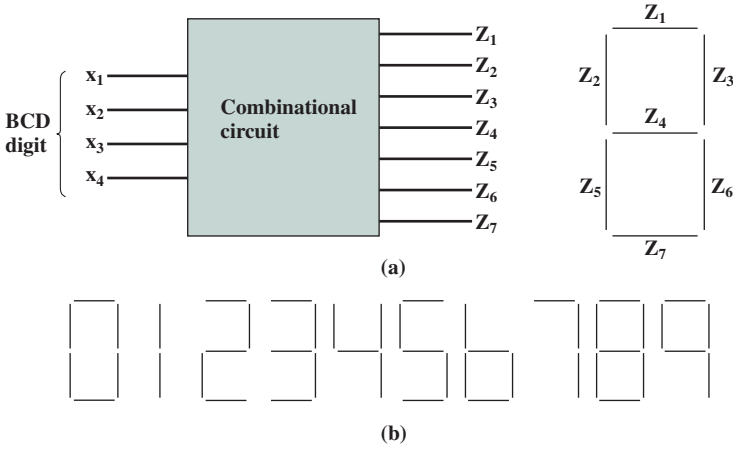


Figure 11.35 Seven-Segment LED Display Example

Binary Code	Gray Code
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

Design a circuit that converts from binary to Gray code.

11.12 Design a 5 × 32 decoder using four 3 × 8 decoders (with enable inputs) and one 2 × 4 decoder.

11.13 Implement the full adder of Figure 11.20 with just five gates. (*Hint:* Some of the gates are **XOR gates**.)

11.14 Consider Figure 11.20. Assume that each gate produces a delay of 10 ns. Thus, the sum output is valid after 20 ns and the carry output after 20 ns. What is the total add time for a 32-bit adder

a. Implemented without carry lookahead, as in Figure 11.19?

b. Implemented with carry lookahead and using 8-bit adders, as in Figure 11.21?

11.15 An alternative form of the S–R latch has the same structure as Figure 11.22 but uses NAND gates instead of NOR gates.

a. Redo Table 11.10a and 11.10b for S–R latch implemented with NAND gates.

b. Complete the following table, similar to Table 11.10c.

t	0	1	2	3	4	5	6	7	8	9
S	0	1	1	1	1	1	0	1	0	1
R	1	1	0	1	0	1	1	1	0	0

11.16 Consider the graphic symbol for the S–R flip-flop in Figure 11.27. Add additional lines to depict a D flip-flop wired from the S–R flip-flop.

11.17 Show the structure of a PLA with three inputs (C, B, A) and four outputs (O_0, O_1, O_2, O_3) with the outputs defined as follows:

$$O_0 = \overline{A}\,\overline{B}C + A\overline{B} + AB\overline{C}$$
$$O_1 = \overline{A}\,\overline{B}C + AB\overline{C}$$
$$O_2 = C$$
$$O_3 = A\overline{B} + AB\overline{C}$$

11.18 An interesting application of a PLA is conversion from the old, obsolete punched card character codes to ASCII codes. The standard punched cards that were so popular with computers in the past had 12 rows and 80 columns where holes could be punched. Each column corresponded to one character, so each character had a 12-bit code. However, only 96 characters were actually used. Consider an application that reads punched cards and converts the character codes to ASCII.

a. Describe a PLA implementation of this application.

b. Can this problem be solved with a ROM? Explain.

PART FOUR THE CENTRAL PROCESSING UNIT

CHAPTER 12

INSTRUCTION SETS: CHARACTERISTICS AND FUNCTIONS

12.1 Machine Instruction Characteristics
- Elements of a Machine Instruction
- Instruction Representation
- Instruction Types
- Number of Addresses
- Instruction Set Design

12.2 Types of Operands
- Numbers
- Characters
- Logical Data

12.3 Intel x86 and ARM Data Types
- x86 Data Types
- ARM Data Types

12.4 Types of Operations
- Data Transfer
- Arithmetic
- Logical
- Conversion
- Input/Output
- System Control
- Transfer of Control

12.5 Intel x86 and ARM Operation Types
- x86 Operation Types
- ARM Operation Types

12.6 Key Terms, Review Questions, and Problems

Appendix 12A Little-, Big-, and Bi-Endian

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Present an overview of essential characteristics of **machine instructions**.
- Describe the types of operands used in typical machine instruction sets.
- Present an overview of x86 and ARM data types.
- Describe the types of operands supported by typical machine instruction sets.
- Present an overview of x86 and ARM operation types.
- Understand the differences among **big endian**, **little endian**, and **bi-endian**.

Much of what is discussed in this book is not readily apparent to the user or programmer of a computer. If a programmer is using a high-level language, such as Pascal or Ada, very little of the architecture of the underlying machine is visible.

One boundary where the computer designer and the computer programmer can view the same machine is the machine instruction set. From the designer's point of view, the machine instruction set provides the functional requirements for the processor: implementing the processor is a task that in large part involves implementing the machine instruction set. The user who chooses to program in machine language (actually, in assembly language; see Appendix B) becomes aware of the register and memory structure, the types of data directly supported by the machine, and the functioning of the ALU.

A description of a computer's machine instruction set goes a long way toward explaining the computer's processor. Accordingly, we focus on machine instructions in this chapter and the next.

12.1 MACHINE INSTRUCTION CHARACTERISTICS

The operation of the processor is determined by the instructions it executes, referred to as *machine instructions* or *computer instructions*. The collection of different instructions that the processor can execute is referred to as the processor's *instruction set*.

Elements of a Machine Instruction

Each instruction must contain the information required by the processor for execution. Figure 12.1, which repeats Figure 3.6, shows the steps involved in instruction execution and, by implication, defines the elements of a machine instruction. These elements are as follows:

- **Operation code:** Specifies the operation to be performed (e.g., ADD, I/O). The operation is specified by a binary code, known as the operation code, or **opcode**.
- **Source operand reference:** The operation may involve one or more source operands, that is, operands that are inputs for the operation.

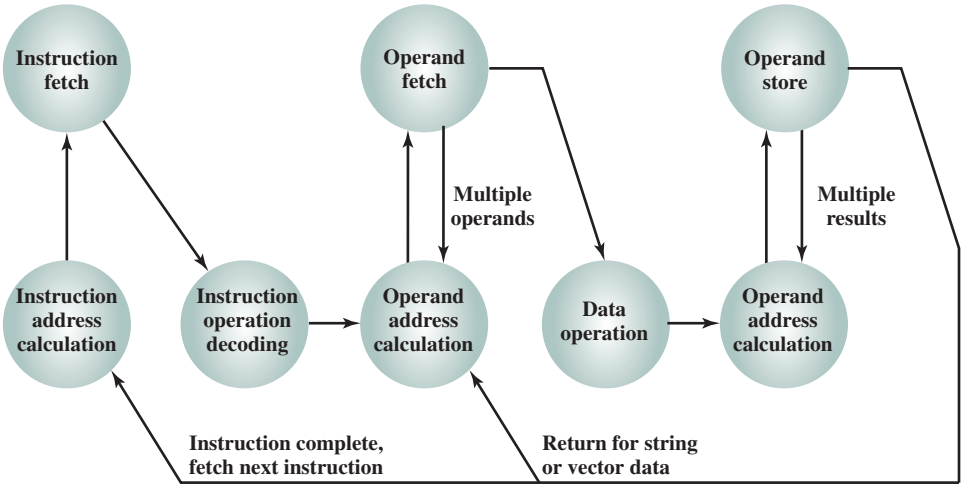


Figure 12.1 Instruction Cycle State Diagram

- **Result operand reference:** The operation may produce a result.
- **Next instruction reference:** This tells the processor where to fetch the next instruction after the execution of this instruction is complete.

The address of the next instruction to be fetched could be either a real address or a virtual address, depending on the architecture. Generally, the distinction is transparent to the instruction set architecture. In most cases, the next instruction to be fetched immediately follows the current instruction. In those cases, there is no explicit reference to the next instruction. When an explicit reference is needed, the main memory or virtual memory address must be supplied. The form in which that address is supplied is discussed in Chapter 13.

Source and result operands can be in one of four areas:

- **Main or virtual memory:** As with next instruction references, the main or virtual memory address must be supplied.
- **Processor register:** With rare exceptions, a processor contains one or more registers that may be referenced by machine instructions. If only one register exists, reference to it may be implicit. If more than one register exists, then each register is assigned a unique name or number, and the instruction must contain the number of the desired register.
- **Immediate:** The value of the operand is contained in a field in the instruction being executed.
- **I/O device:** The instruction must specify the I/O module and device for the operation. If memory-mapped I/O is used, this is just another main or virtual memory address.

Instruction Representation

Within the computer, each instruction is represented by a sequence of bits. The instruction is divided into fields, corresponding to the constituent elements of the

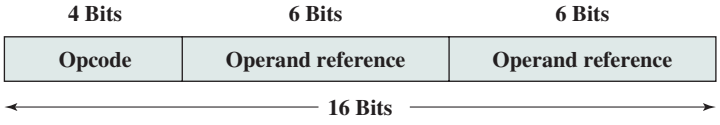


Figure 12.2 A Simple Instruction Format

instruction. A simple example of an instruction format is shown in Figure 12.2. As another example, the IAS instruction format is shown in Figure 2.2. With most instruction sets, more than one format is used. During instruction execution, an instruction is read into an instruction register (IR) in the processor. The processor must be able to extract the data from the various instruction fields to perform the required operation.

It is difficult for both the programmer and the reader of textbooks to deal with binary representations of machine instructions. Thus, it has become common practice to use a *symbolic representation* of machine instructions. An example of this was used for the IAS instruction set, in Table 1.1.

Opcodes are represented by abbreviations, called *mnemonics,* that indicate the operation. Common examples include

ADD	Add
SUB	Subtract
MUL	Multiply
DIV	Divide
LOAD	Load data from memory
STOR	Store data to memory

Operands are also represented symbolically. For example, the instruction

ADD R, Y

may mean add the value contained in data location Y to the contents of register R. In this example, Y refers to the address of a location in memory, and R refers to a particular register. Note that the operation is performed on the contents of a location, not on its address.

Thus, it is possible to write a machine-language program in symbolic form. Each symbolic opcode has a fixed binary representation, and the programmer specifies the location of each symbolic operand. For example, the programmer might begin with a list of definitions:

X = 513
Y = 514

and so on. A simple program would accept this symbolic input, convert opcodes and operand references to binary form, and construct binary machine instructions.

Machine-language programmers are rare to the point of nonexistence. Most programs today are written in a high-level language or, failing that, assembly language, which is discussed in Appendix B. However, symbolic machine language remains a useful tool for describing machine instructions, and we will use it for that purpose.

Instruction Types

Consider a high-level language instruction that could be expressed in a language such as BASIC or FORTRAN. For example,

$$X = X + Y$$

This statement instructs the computer to add the value stored in Y to the value stored in X and put the result in X. How might this be accomplished with machine instructions? Let us assume that the variables X and Y correspond to locations 513 and 514. If we assume a simple set of machine instructions, this operation could be accomplished with three instructions:

1. Load a register with the contents of memory location 513.
2. Add the contents of memory location 514 to the register.
3. Store the contents of the register in memory location 513.

As can be seen, the single BASIC instruction may require three machine instructions. This is typical of the relationship between a high-level language and a machine language. A high-level language expresses operations in a concise algebraic form, using variables. A machine language expresses operations in a basic form involving the movement of data to or from registers.

With this simple example to guide us, let us consider the types of instructions that must be included in a practical computer. A computer should have a set of instructions that allows the user to formulate any data processing task. Another way to view it is to consider the capabilities of a high-level programming language. Any program written in a high-level language must be translated into machine language to be executed. Thus, the set of machine instructions must be sufficient to express any of the instructions from a high-level language. With this in mind we can categorize instruction types as follows:

- **Data processing:** Arithmetic and logic instructions.
- **Data storage:** Movement of data into or out of register and or memory locations.
- **Data movement:** I/O instructions.
- **Control:** Test and branch instructions.

Arithmetic instructions provide computational capabilities for processing numeric data. *Logic* (Boolean) instructions operate on the bits of a word as bits rather than as numbers; thus, they provide capabilities for processing any other type of data the user may wish to employ. These operations are performed primarily on data in processor registers. Therefore, there must be *memory* instructions for moving data between memory and the registers. *I/O* instructions are needed to transfer programs and data into memory and the results of computations back out to the user. *Test* instructions are used to test the value of a data word or the status of a computation. *Branch* instructions are then used to branch to a different set of instructions depending on the decision made.

We will examine the various types of instructions in greater detail later in this chapter.

Number of Addresses

One of the traditional ways of describing processor architecture is in terms of the number of addresses contained in each instruction. This dimension has become less significant with the increasing complexity of processor design. Nevertheless, it is useful at this point to draw and analyze this distinction.

What is the maximum number of addresses one might need in an instruction? Evidently, arithmetic and logic instructions will require the most operands. Virtually all arithmetic and logic operations are either unary (one source operand) or binary (two source operands). Thus, we would need a maximum of two addresses to reference source operands. The result of an operation must be stored, suggesting a third address, which defines a destination operand. Finally, after completion of an instruction, the next instruction must be fetched, and its address is needed.

This line of reasoning suggests that an instruction could plausibly be required to contain four address references: two source operands, one destination operand, and the address of the next instruction. In most architectures, many instructions have one, two, or three operand addresses, with the address of the next instruction being implicit (obtained from the program counter). Most architectures also have a few special-purpose instructions with more operands. For example, the load and store multiple instructions of the ARM architecture, described in Chapter 13, designate up to 17 register operands in a single instruction.

Figure 12.3 compares typical one-, two-, and three-address instructions that could be used to compute $Y = (A - B)/[C + (D \times E)]$. With three addresses, each instruction specifies two source operand locations and a destination operand location. Because we choose not to alter the value of any of the operand locations,

Instruction		Comment
SUB	Y, A, B	Y ← A − B
MPY	T, D, E	T ← D × E
ADD	T, T, C	T ← T + C
DIV	Y, Y, T	Y ← Y ÷ T

(a) Three-address instructions

Instruction		Comment
MOVE	Y, A	Y ← A
SUB	Y, B	Y ← Y − B
MOVE	T, D	T ← D
MPY	T, E	T ← T × E
ADD	T, C	T ← T + C
DIV	Y, T	Y ← Y ÷ T

(b) Two-address instructions

Instruction		Comment
LOAD	D	AC ← D
MPY	E	AC ← AC × E
ADD	C	AC ← AC + C
STOR	Y	Y ← AC
LOAD	A	AC ← A
SUB	B	AC ← AC − B
DIV	Y	AC ← AC ÷ Y
STOR	Y	Y ← AC

(c) One-address instructions

Figure 12.3 Programs to Execute $Y = \dfrac{A - B}{C + (D \times E)}$

a temporary location, T, is used to store some intermediate results. Note that there are four instructions and that the original expression had five operands.

Three-address instruction formats are not common because they require a relatively long instruction format to hold the three address references. With two-address instructions, and for binary operations, one address must do double duty as both an operand and a result. Thus, the instruction SUB Y, B carries out the calculation Y − B and stores the result in Y. The two-address format reduces the space requirement but also introduces some awkwardness. To avoid altering the value of an operand, a MOVE instruction is used to move one of the values to a result or temporary location before performing the operation. Our sample program expands to six instructions.

Simpler yet is the one-address instruction. For this to work, a second address must be implicit. This was common in earlier machines, with the implied address being a processor register known as the **accumulator** (AC). The accumulator contains one of the operands and is used to store the result. In our example, eight instructions are needed to accomplish the task.

It is, in fact, possible to make do with zero addresses for some instructions. Zero-address instructions are applicable to a special memory organization called a stack. A stack is a last-in-first-out set of locations. The stack is in a known location and, often, at least the top two elements are in processor registers. Thus, zero-address instructions would reference the top two stack elements. Stacks are described in Appendix I. Their use is explored further later in this chapter and in Chapter 13.

Table 12.1 summarizes the interpretations to be placed on instructions with zero, one, two, or three addresses. In each case in the table, it is assumed that the address of the next instruction is implicit, and that one operation with two source operands and one result operand is to be performed.

The number of addresses per instruction is a basic design decision. Fewer addresses per instruction result in instructions that are more primitive, requiring a less complex processor. It also results in instructions of shorter length. On the other hand, programs contain more total instructions, which in general results in longer execution times and longer, more complex programs. Also, there is an important threshold between one-address and multiple-address instructions. With one-address instructions, the programmer generally has available only one general-purpose

Table 12.1 Utilization of Instruction Addresses (Nonbranching Instructions)

Number of Addresses	Symbolic Representation	Interpretation
3	OP A, B, C	A ← B OP C
2	OP A, B	A ← A OP B
1	OP A	AC ← AC OP A
0	OP	T ← (T − 1) OP T

AC = accumulator
T = top of stack
(T − 1) = second element of stack
A, B, C = memory or register locations

register, the accumulator. With multiple-address instructions, it is common to have multiple general-purpose registers. This allows some operations to be performed solely on registers. Because register references are faster than memory references, this speeds up execution. For reasons of flexibility and ability to use multiple registers, most contemporary machines employ a mixture of two- and three-address instructions.

The design trade-offs involved in choosing the number of addresses per instruction are complicated by other factors. There is the issue of whether an address references a memory location or a register. Because there are fewer registers, fewer bits are needed for a register reference. Also, as we will see in Chapter 13, a machine may offer a variety of addressing modes, and the specification of mode takes one or more bits. The result is that most processor designs involve a variety of instruction formats.

Instruction Set Design

One of the most interesting, and most analyzed, aspects of computer design is instruction set design. The design of an instruction set is very complex because it affects so many aspects of the computer system. The instruction set defines many of the functions performed by the processor and thus has a significant effect on the implementation of the processor. The instruction set is the programmer's means of controlling the processor. Thus, programmer requirements must be considered in designing the instruction set.

It may surprise you to know that some of the most fundamental issues relating to the design of instruction sets remain in dispute. Indeed, in recent years, the level of disagreement concerning these fundamentals has actually grown. The most important of these fundamental design issues include the following:

- **Operation repertoire:** How many and which operations to provide, and how complex operations should be.
- **Data types:** The various types of data upon which operations are performed.
- **Instruction format:** Instruction length (in bits), number of addresses, size of various fields, and so on.
- **Registers:** Number of processor registers that can be referenced by instructions, and their use.
- **Addressing:** The mode or modes by which the address of an operand is specified.

These issues are highly interrelated and must be considered together in designing an instruction set. This book, of course, must consider them in some sequence, but an attempt is made to show the interrelationships.

Because of the importance of this topic, much of Part Three is devoted to instruction set design. Following this overview section, this chapter examines data types and operation repertoire. Chapter 13 examines addressing modes (which includes a consideration of registers) and instruction formats. Chapter 15 examines the reduced instruction set computer (RISC). RISC architecture calls into question many of the instruction set design decisions traditionally made in commercial computers.

12.2 TYPES OF OPERANDS

Machine instructions operate on data. The most important general categories of data are

- Addresses
- Numbers
- Characters
- Logical data

We shall see, in discussing addressing modes in Chapter 13, that addresses are, in fact, a form of data. In many cases, some calculation must be performed on the operand reference in an instruction to determine the main or virtual memory address. In this context, addresses can be considered to be unsigned integers.

Other common data types are numbers, characters, and logical data, and each of these is briefly examined in this section. Beyond that, some machines define specialized data types or data structures. For example, there may be machine operations that operate directly on a list or a string of characters.

Numbers

All machine languages include numeric data types. Even in nonnumeric data processing, there is a need for numbers to act as counters, field widths, and so forth. An important distinction between numbers used in ordinary mathematics and numbers stored in a computer is that the latter are limited. This is true in two senses. First, there is a limit to the magnitude of numbers representable on a machine and second, in the case of floating-point numbers, a limit to their precision. Thus, the programmer is faced with understanding the consequences of rounding, overflow, and underflow.

Three types of numerical data are common in computers:

- Binary integer or binary fixed point
- Binary floating point
- Decimal

We examined the first two in some detail in Chapter 10. It remains to say a few words about decimal numbers.

Although all internal computer operations are binary in nature, the human users of the system deal with decimal numbers. Thus, there is a necessity to convert from decimal to binary on input and from binary to decimal on output. For applications in which there is a great deal of I/O and comparatively little, comparatively simple computation, it is preferable to store and operate on the numbers in decimal form. The most common representation for this purpose is **packed decimal**.[1]

[1]Textbooks often refer to this as binary coded decimal (BCD). Strictly speaking, BCD refers to the encoding of each decimal digit by a unique 4-bit sequence. Packed decimal refers to the storage of BCD-encoded digits using one byte for each two digits.

With packed decimal, each decimal digit is represented by a 4-bit code, in the obvious way, with two digits stored per byte. Thus, 0 = 000, 1 = 0001, . . . , 8 = 1000, and 9 = 1001. Note that this is a rather inefficient code because only 10 of 16 possible 4-bit values are used. To form numbers, 4-bit codes are strung together, usually in multiples of 8 bits. Thus, the code for 246 is 0000 0010 0100 0110. This code is clearly less compact than a straight binary representation, but it avoids the conversion overhead. Negative numbers can be represented by including a 4-bit sign digit at either the left or right end of a string of packed decimal digits. Standard sign values are 1100 for positive (+) and 1101 for negative (−).

Many machines provide arithmetic instructions for performing operations directly on packed decimal numbers. The algorithms are quite similar to those described in Section 9.3 but must take into account the decimal carry operation.

Characters

A common form of data is text or character strings. While textual data are most convenient for human beings, they cannot, in character form, be easily stored or transmitted by data processing and communications systems. Such systems are designed for binary data. Thus, a number of codes have been devised by which characters are represented by a sequence of bits. Perhaps the earliest common example of this is the Morse code. Today, the most commonly used character code in the International Reference Alphabet (IRA), referred to in the United States as the American Standard Code for Information Interchange (ASCII; see Appendix H). Each character in this code is represented by a unique 7-bit pattern; thus, 128 different characters can be represented. This is a larger number than is necessary to represent printable characters, and some of the patterns represent *control* characters. Some of these control characters have to do with controlling the printing of characters on a page. Others are concerned with communications procedures. IRA-encoded characters are almost always stored and transmitted using 8 bits per character. The eighth bit may be set to 0 or used as a parity bit for error detection. In the latter case, the bit is set such that the total number of binary 1s in each octet is always odd (odd parity) or always even (even parity).

Note in Table H.1 (Appendix H) that for the IRA bit pattern 011XXXX, the digits 0 through 9 are represented by their binary equivalents, 0000 through 1001, in the rightmost 4 bits. This is the same code as packed decimal. This facilitates conversion between 7-bit IRA and 4-bit packed decimal representation.

Another code used to encode characters is the Extended Binary Coded Decimal Interchange Code (EBCDIC). EBCDIC is used on IBM mainframes. It is an 8-bit code. As with IRA, EBCDIC is compatible with packed decimal. In the case of EBCDIC, the codes 11110000 through 11111001 represent the digits 0 through 9.

Logical Data

Normally, each word or other addressable unit (byte, halfword, and so on) is treated as a single unit of data. It is sometimes useful, however, to consider an n-bit unit as consisting of n 1-bit items of data, each item having the value 0 or 1. When data are viewed this way, they are considered to be *logical* data.

There are two advantages to the bit-oriented view. First, we may sometimes wish to store an array of Boolean or binary data items, in which each item can take on only the values 1 (true) and 0 (false). With logical data, memory can be used most efficiently for this storage. Second, there are occasions when we wish to manipulate the bits of a data item. For example, if floating-point operations are implemented in software, we need to be able to shift significant bits in some operations. Another example: To convert from IRA to packed decimal, we need to extract the rightmost 4 bits of each byte.

Note that, in the preceding examples, the same data are treated sometimes as logical and other times as numerical or text. The "type" of a unit of data is determined by the operation being performed on it. While this is not normally the case in high-level languages, it is almost always the case with machine language.

12.3 INTEL x86 AND ARM DATA TYPES

x86 Data Types

The x86 can deal with data types of 8 (byte), 16 (word), 32 (doubleword), 64 (quadword), and 128 (double quadword) bits in length. To allow maximum flexibility in data structures and efficient memory utilization, words need not be aligned at even-numbered addresses; doublewords need not be aligned at addresses evenly divisible by 4; quadwords need not be aligned at addresses evenly divisible by 8; and so on. However, when data are accessed across a 32-bit bus, data transfers take place in units of doublewords, beginning at addresses divisible by 4. The processor converts the request for misaligned values into a sequence of requests for the bus transfer. As with all of the Intel 80x86 machines, the x86 uses the little-endian style; that is, the least significant byte is stored in the lowest address (see Appendix 12A for a discussion of endianness).

The byte, word, doubleword, quadword, and double quadword are referred to as general data types. In addition, the x86 supports an impressive array of specific data types that are recognized and operated on by particular instructions. Table 12.2 summarizes these types.

Figure 12.4 illustrates the x86 numerical data types. The signed integers are in twos complement representation and may be 16, 32, or 64 bits long. The floating-point type actually refers to a set of types that are used by the floating-point unit and operated on by floating-point instructions. The floating-point representations conform to the IEEE 754 standard.

The packed SIMD (single-instruction-multiple-data) data types were introduced to the x86 architecture as part of the extensions of the instruction set to optimize performance of multimedia applications. These extensions include MMX (multimedia extensions) and SSE (streaming SIMD extensions). The basic concept is that multiple operands are packed into a single referenced memory item and that these multiple operands are operated on in parallel. The data types are as follows:

- **Packed byte and packed byte integer:** Bytes packed into a 64-bit quadword or 128-bit double quadword, interpreted as a bit field or as an integer.
- **Packed word and packed word integer:** 16-bit words packed into a 64-bit quadword or 128-bit double quadword, interpreted as a bit field or as an integer.

Table 12.2 x86 Data Types

Data Type	Description
General	Byte, word (16 bits), doubleword (32 bits), quadword (64 bits), and double quadword (128 bits) locations with arbitrary binary contents.
Integer	A signed binary value contained in a byte, word, or doubleword, using twos complement representation.
Ordinal	An unsigned integer contained in a byte, word, or doubleword.
Unpacked binary coded decimal (BCD)	A representation of a BCD digit in the range 0 through 9, with one digit in each byte.
Packed BCD	Packed byte representation of two BCD digits; value in the range 0 to 99.
Near pointer	A 16-bit, 32-bit, or 64-bit effective address that represents the offset within a segment. Used for all pointers in a nonsegmented memory and for references within a segment in a segmented memory.
Far pointer	A logical address consisting of a 16-bit segment selector and an offset of 16, 32, or 64 bits. Far pointers are used for memory references in a segmented memory model where the identity of a segment being accessed must be specified explicitly.
Bit field	A contiguous sequence of bits in which the position of each bit is considered as an independent unit. A bit string can begin at any bit position of any byte and can contain up to 32 bits.
Bit string	A contiguous sequence of bits, containing from zero to $2^{23} - 1$ bits.
Byte string	A contiguous sequence of bytes, words, or doublewords, containing from zero to $2^{23} - 1$ bytes.
Floating point	See Figure 12.4.
Packed SIMD (single instruction, multiple data)	Packed 64-bit and 128-bit data types.

- **Packed doubleword and packed doubleword integer:** 32-bit doublewords packed into a 64-bit quadword or 128-bit double quadword, interpreted as a bit field or as an integer.
- **Packed quadword and packed quadword integer:** Two 64-bit quadwords packed into a 128-bit double quadword, interpreted as a bit field or as an integer.
- **Packed single-precision floating-point and packed double-precision floating-point:** Four 32-bit floating-point or two 64-bit floating-point values packed into a 128-bit double quadword.

ARM Data Types

ARM processors support data types of 8 (byte), 16 (halfword), and 32 (word) bits in length. Normally, halfword access should be halfword aligned and word accesses should be word aligned. For nonaligned access attempts, the architecture supports three alternatives.

- Default case:
  - The address is treated as truncated, with address bits[1:0] treated as zero for word accesses, and address bit[0] treated as zero for halfword accesses.

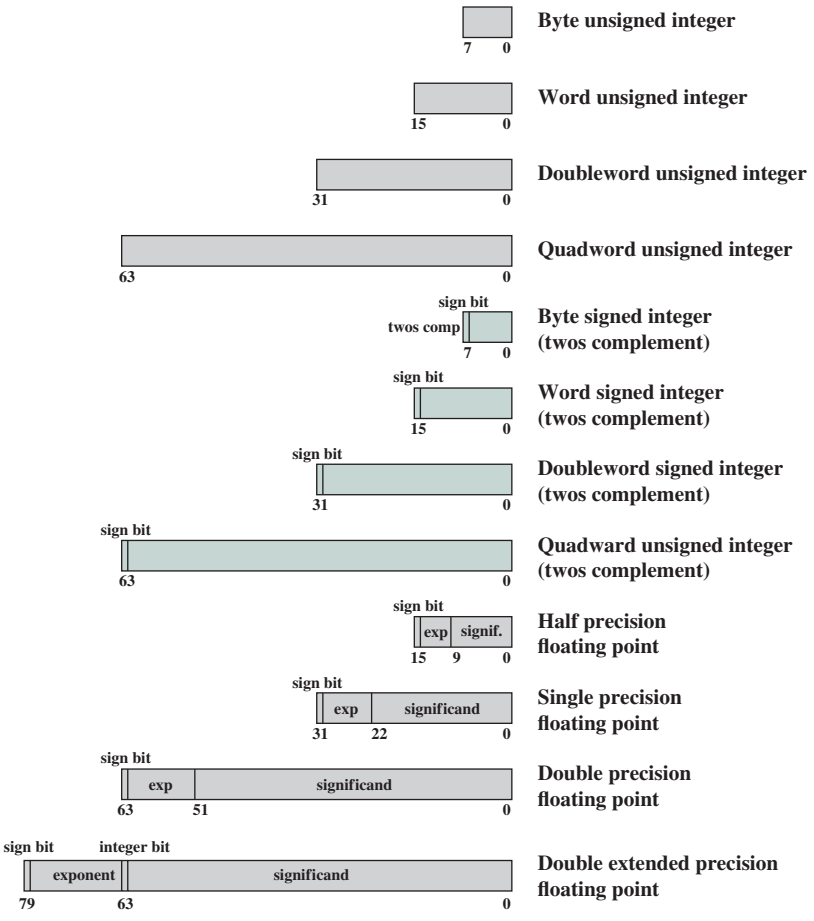


Figure 12.4 x86 Numeric Data Formats

– Load single word ARM instructions are architecturally defined to rotate right the word-aligned data transferred by a non word-aligned address one, two, or three bytes depending on the value of the two least significant address bits.

- **Alignment checking:** When the appropriate control bit is set, a data abort signal indicates an alignment fault for attempting unaligned access.
- **Unaligned access:** When this option is enabled, the processor uses one or more memory accesses to generate the required transfer of adjacent bytes transparently to the programmer.

For all three data types (byte, halfword, and word) an unsigned interpretation is supported, in which the value represents an unsigned, nonnegative integer. All three data types can also be used for twos complement signed integers.

The majority of ARM processor implementations do not provide floating-point hardware, which saves power and area. If floating-point arithmetic is required in such processors, it must be implemented in software. ARM does support an optional floating-point coprocessor that supports the single- and double-precision floating point data types defined in IEEE 754.

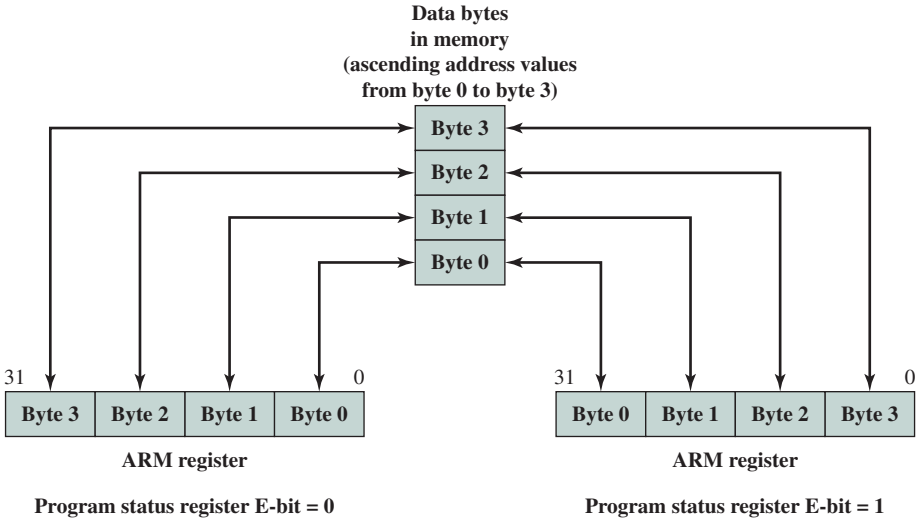


Figure 12.5 ARM Endian Support—Word Load/Store with E-Bit

ENDIAN SUPPORT A state bit (E-bit) in the system control register is set and cleared under program control using the SETEND instruction. The E-bit defines which endian to load and store data. Figure 12.5 illustrates the functionality associated with the E-bit for a word load or store operation. This mechanism enables efficient dynamic data load/store for system designers who know they need to access data structures in the opposite endianness to their OS/environment. Note that the address of each data byte is fixed in memory. However, the byte lane in a register is different.

12.4 TYPES OF OPERATIONS

The number of different opcodes varies widely from machine to machine. However, the same general types of operations are found on all machines. A useful and typical categorization is the following:

- Data transfer
- Arithmetic
- Logical
- Conversion
- I/O
- System control
- Transfer of control

Table 12.3 (based on [HAYE98]) lists common instruction types in each category. This section provides a brief survey of these various types of operations, together with a brief discussion of the actions taken by the processor to execute a particular type of operation (summarized in Table 12.4). The latter topic is examined in more detail in Chapter 14.

Table 12.3 Common Instruction Set Operations

Type	Operation Name	Description
Data transfer	Move (transfer)	Transfer word or block from source to destination
	Store	Transfer word from processor to memory
	Load (fetch)	Transfer word from memory to processor
	Exchange	Swap contents of source and destination
	Clear (reset)	Transfer word of 0s to destination
	Set	Transfer word of 1s to destination
	Push	Transfer word from source to top of stack
	Pop	Transfer word from top of stack to destination
Arithmetic	Add	Compute sum of two operands
	Subtract	Compute difference of two operands
	Multiply	Compute product of two operands
	Divide	Compute quotient of two operands
	Absolute	Replace operand by its absolute value
	Negate	Change sign of operand
	Increment	Add 1 to operand
	Decrement	Subtract 1 from operand
Logical	AND	Perform logical AND
	OR	Perform logical OR
	NOT	(complement) Perform logical NOT
	Exclusive-OR	Perform logical XOR
	Test	Test specified condition; set flag(s) based on outcome
	Compare	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome
	Set Control Variables	Class of instructions to set controls for protection purposes, interrupt handling, timer control, etc.
	Shift	Left (right) shift operand, introducing constants at end
	Rotate	Left (right) shift operand, with wraparound end
Transfer of control	Jump (branch)	Unconditional transfer; load PC with specified address
	Jump Conditional	Test specified condition; either load PC with specified address or do nothing, based on condition
	Jump to Subroutine	Place current program control information in known location; jump to specified address
	Return	Replace contents of PC and other register from known location
	Execute	Fetch operand from specified location and execute as instruction; do not modify PC
	Skip	Increment PC to skip next instruction
	Skip Conditional	Test specified condition; either skip or do nothing based on condition
	Halt	Stop program execution
	Wait (hold)	Stop program execution; test specified condition repeatedly; resume execution when condition is satisfied
	No operation	No operation is performed, but program execution is continued

Type	Operation Name	Description
Input/output	Input (read)	Transfer data from specified I/O port or device to destination (e.g., main memory or processor register)
	Output (write)	Transfer data from specified source to I/O port or device
	Start I/O	Transfer instructions to I/O processor to initiate I/O operation
	Test I/O	Transfer status information from I/O system to specified destination
Conversion	Translate	Translate values in a section of memory based on a table of correspondences
	Convert	Convert the contents of a word from one form to another (e.g., packed decimal to binary)

Table 12.4 Processor Actions for Various Types of Operations

Data transfer	Transfer data from one location to another
	If memory is involved: Determine memory address Perform virtual-to-actual-memory address transformation Check cache Initiate memory read/write
Arithmetic	May involve data transfer, before and/or after
	Perform function in ALU
	Set condition codes and flags
Logical	Same as arithmetic
Conversion	Similar to arithmetic and logical. May involve special logic to perform conversion
Transfer of control	Update program counter. For subroutine call/return, manage parameter passing and linkage
I/O	Issue command to I/O module
	If memory-mapped I/O, determine memory-mapped address

Data Transfer

The most fundamental type of machine instruction is the data transfer instruction. The data transfer instruction must specify several things. First, the location of the source and destination operands must be specified. Each location could be memory, a register, or the top of the stack. Second, the length of data to be transferred must be indicated. Third, as with all instructions with operands, the mode of addressing for each operand must be specified. This latter point is discussed in Chapter 13.

The choice of data transfer instructions to include in an instruction set exemplifies the kinds of trade-offs the designer must make. For example, the general location (memory or register) of an operand can be indicated in either the specification of the opcode or the operand. Table 12.5 shows examples of the most common IBM EAS/390 data transfer instructions. Note that there are variants to indicate

Table 12.5 Examples of IBM EAS/390 Data Transfer Operations

Operation Mnemonic	Name	Number of Bits Transferred	Description
L	Load	32	Transfer from memory to register
LH	Load Halfword	16	Transfer from memory to register
LR	Load	32	Transfer from register to register
LER	Load (short)	32	Transfer from floating-point register to floating-point register
LE	Load (short)	32	Transfer from memory to floating-point register
LDR	Load (long)	64	Transfer from floating-point register to floating-point register
LD	Load (long)	64	Transfer from memory to floating-point register
ST	Store	32	Transfer from register to memory
STH	Store Halfword	16	Transfer from register to memory
STC	Store Character	8	Transfer from register to memory
STE	Store (short)	32	Transfer from floating-point register to memory
STD	Store (long)	64	Transfer from floating-point register to memory

the amount of data to be transferred (8, 16, 32, or 64 bits). Also, there are different instructions for register to register, register to memory, memory to register, and memory to memory transfers. In contrast, the VAX has a move (MOV) instruction with variants for different amounts of data to be moved, but it specifies whether an operand is register or memory as part of the operand. The VAX approach is somewhat easier for the programmer, who has fewer mnemonics to deal with. However, it is also somewhat less compact than the IBM EAS/390 approach because the location (register versus memory) of each operand must be specified separately in the instruction. We will return to this distinction when we discuss instruction formats in Chapter 13.

In terms of processor action, data transfer operations are perhaps the simplest type. If both source and destination are registers, then the processor simply causes data to be transferred from one register to another; this is an operation internal to the processor. If one or both operands are in memory, then the processor must perform some or all of the following actions:

1. Calculate the memory address, based on the address mode (discussed in Chapter 13).
2. If the address refers to virtual memory, translate from virtual to real memory address.
3. Determine whether the addressed item is in cache.
4. If not, issue a command to the memory module.

Arithmetic

Most machines provide the basic arithmetic operations of add, subtract, multiply, and divide. These are invariably provided for signed integer (fixed-point) numbers. Often they are also provided for floating-point and packed decimal numbers.

Other possible operations include a variety of single-operand instructions; for example,

- **Absolute:** Take the absolute value of the operand.
- **Negate:** Negate the operand.
- **Increment:** Add 1 to the operand.
- **Decrement:** Subtract 1 from the operand.

The execution of an arithmetic instruction may involve data transfer operations to position operands for input to the ALU, and to deliver the output of the ALU. Figure 3.5 illustrates the movements involved in both data transfer and arithmetic operations. In addition, of course, the ALU portion of the processor performs the desired operation.

Logical

Most machines also provide a variety of operations for manipulating individual bits of a word or other addressable units, often referred to as "bit twiddling." They are based upon Boolean operations (see Chapter 11).

Some of the basic logical operations that can be performed on Boolean or binary data are shown in Table 12.6. The NOT operation inverts a bit. AND, OR, and Exclusive-OR (XOR) are the most common logical functions with two operands. EQUAL is a useful binary test.

These logical operations can be applied bitwise to n-bit logical data units. Thus, if two registers contain the data

$$(R1) = 10100101$$
$$(R2) = 00001111$$

then

$$(R1) \text{ AND } (R2) = 00000101$$

Table 12.6 Basic Logical Operations

P	Q	NOT P	P AND Q	P OR Q	P XOR Q	P = Q
0	0	1	0	0	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	0
1	1	0	1	1	0	1

where the notation (X) means the contents of location X. Thus, the AND operation can be used as a *mask* that selects certain bits in a word and zeros out the remaining bits. As another example, if two registers contain

$$(R1) = 10100101$$
$$(R2) = 11111111$$

then

$$(R1) \text{ XOR } (R2) = 01011010$$

With one word set to all 1s, the XOR operation inverts all of the bits in the other word (ones complement).

In addition to bitwise logical operations, most machines provide a variety of shifting and rotating functions. The most basic operations are illustrated in Figure 12.6. With a **logical shift**, the bits of a word are shifted left or right. On one end, the bit shifted out is lost. On the other end, a 0 is shifted in. Logical shifts are useful primarily for isolating fields within a word. The 0s that are shifted into a word displace unwanted information that is shifted off the other end.

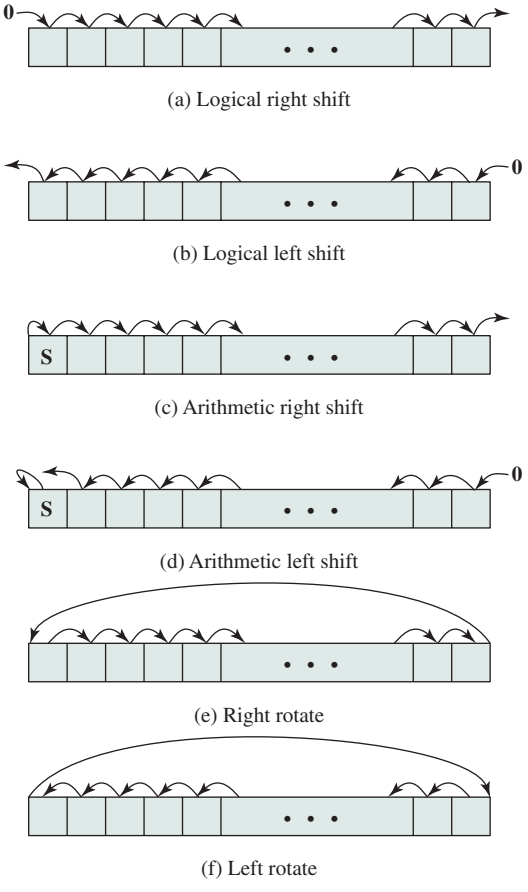


Figure 12.6 Shift and Rotate Operations

As an example, suppose we wish to transmit characters of data to an I/O device 1 character at a time. If each memory word is 16 bits in length and contains two characters, we must *unpack* the characters before they can be sent. To send the two characters in a word;

1. Load the word into a register.
2. Shift to the right eight times. This shifts the remaining character to the right half of the register.
3. Perform I/O. The I/O module reads the lower-order 8 bits from the data bus.

The preceding steps result in sending the left-hand character. To send the right-hand character;

1. Load the word again into the register.
2. AND with 0000000011111111. This masks out the character on the left.
3. Perform I/O.

The **arithmetic shift** operation treats the data as a signed integer and does not shift the sign bit. On a right arithmetic shift, the sign bit is replicated into the bit position to its right. On a left arithmetic shift, a logical left shift is performed on all bits but the sign bit, which is retained. These operations can speed up certain arithmetic operations. With numbers in twos complement notation, a right arithmetic shift corresponds to a division by 2, with truncation for odd numbers. Both an arithmetic left shift and a logical left shift correspond to a multiplication by 2 when there is no overflow. If overflow occurs, arithmetic and logical left shift operations produce different results, but the arithmetic left shift retains the sign of the number. Because of the potential for overflow, many processors do not include this instruction, including PowerPC and Itanium. Others, such as the IBM EAS/390, do offer the instruction. Curiously, the x86 architecture includes an arithmetic left shift but defines it to be identical to a logical left shift.

Rotate, or cyclic shift, operations preserve all of the bits being operated on. One use of a rotate is to bring each bit successively into the leftmost bit, where it can be identified by testing the sign of the data (treated as a number).

As with arithmetic operations, logical operations involve ALU activity and may involve data transfer operations. Table 12.7 gives examples of all of the shift and rotate operations discussed in this subsection.

Table 12.7 Examples of Shift and Rotate Operations

Input	Operation	Result
10100110	Logical right shift (3 bits)	00010100
10100110	Logical left shift (3 bits)	00110000
10100110	Arithmetic right shift (3 bits)	11110100
10100110	Arithmetic left shift (3 bits)	10110000
10100110	Right rotate (3 bits)	11010100
10100110	Left rotate (3 bits)	00110101

Conversion

Conversion instructions are those that change the format or operate on the format of data. An example is converting from decimal to binary. An example of a more complex editing instruction is the EAS/390 Translate (TR) instruction. This instruction can be used to convert from one 8-bit code to another, and it takes three operands:

TR R1 (L), R2

The operand R2 contains the address of the start of a table of 8-bit codes. The L bytes starting at the address specified in R1 are translated, each byte being replaced by the contents of a table entry indexed by that byte. For example, to translate from EBCDIC to IRA, we first create a 256-byte table in storage locations, say, 1000-10FF hexadecimal. The table contains the characters of the IRA code in the sequence of the binary representation of the EBCDIC code; that is, the IRA code is placed in the table at the relative location equal to the binary value of the EBCDIC code of the same character. Thus, locations 10F0 through 10F9 will contain the values 30 through 39, because F0 is the EBCDIC code for the digit 0, and 30 is the IRA code for the digit 0, and so on through digit 9. Now suppose we have the EBCDIC for the digits 1984 starting at location 2100 and we wish to translate to IRA. Assume the following:

- Locations 2100–2103 contain F1 F9 F8 F4.
- R1 contains 2100.
- R2 contains 1000.

Then, if we execute

TR R1 (4), R2

locations 2100–2103 will contain 31 39 38 34.

Input/Output

Input/output instructions were discussed in some detail in Chapter 7. As we saw, there are a variety of approaches taken, including isolated programmed I/O, memory-mapped programmed I/O, DMA, and the use of an I/O processor. Many implementations provide only a few I/O instructions, with the specific actions specified by parameters, codes, or command words.

System Control

System control instructions are those that can be executed only while the processor is in a certain privileged state or is executing a program in a special privileged area of memory. Typically, these instructions are reserved for the use of the operating system.

Some examples of system control operations are as follows. A system control instruction may read or alter a control register; we discuss control registers in Chapter 14. Another example is an instruction to read or modify a storage protection key, such as is used in the EAS/390 memory system. Yet another example is access to process control blocks in a multiprogramming system.

Transfer of Control

For all of the operation types discussed so far, the next instruction to be performed is the one that immediately follows, in memory, the current instruction. However, a significant fraction of the instructions in any program have as their function changing the sequence of instruction execution. For these instructions, the operation performed by the processor is to update the program counter to contain the address of some instruction in memory.

There are a number of reasons why transfer-of-control operations are required. Among the most important are the following:

1. In the practical use of computers, it is essential to be able to execute each instruction more than once and perhaps many thousands of times. It may require thousands or perhaps millions of instructions to implement an application. This would be unthinkable if each instruction had to be written out separately. If a table or a list of items is to be processed, a program loop is needed. One sequence of instructions is executed repeatedly to process all the data.
2. Virtually all programs involve some decision making. We would like the computer to do one thing if one condition holds, and another thing if another condition holds. For example, a sequence of instructions computes the square root of a number. At the start of the sequence, the sign of the number is tested. If the number is negative, the computation is not performed, but an error condition is reported.
3. To compose correctly a large or even medium-size computer program is an exceedingly difficult task. It helps if there are mechanisms for breaking the task up into smaller pieces that can be worked on one at a time.

We now turn to a discussion of the most common transfer-of-control operations found in instruction sets: branch, **skip**, and **procedure call**.

BRANCH INSTRUCTIONS A branch instruction, also called a jump instruction, has as one of its operands the address of the next instruction to be executed. Most often, the instruction is a **conditional branch** instruction. That is, the branch is made (update program counter to equal address specified in operand) only if a certain condition is met. Otherwise, the next instruction in sequence is executed (increment program counter as usual). A branch instruction in which the branch is always taken is an **unconditional branch**.

There are two common ways of generating the condition to be tested in a conditional branch instruction. First, most machines provide a 1-bit or multiple-bit condition code that is set as the result of some operations. This code can be thought of as a short user-visible register. As an example, an arithmetic operation (ADD, SUBTRACT, and so on) could set a 2-bit condition code with one of the following four values: 0, positive, negative, overflow. On such a machine, there could be four different conditional branch instructions:

BRP X	Branch to location X if result is positive.
BRN X	Branch to location X if result is negative.
BRZ X	Branch to location X if result is zero.
BRO X	Branch to location X if overflow occurs.

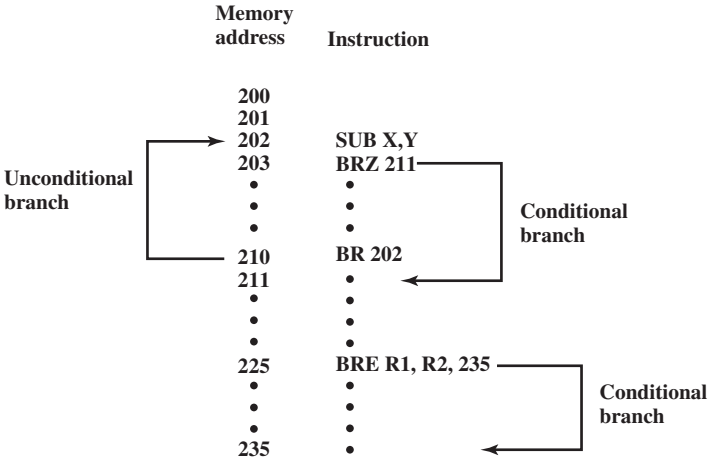


Figure 12.7 Branch Instructions

In all of these cases, the result referred to is the result of the most recent operation that set the condition code.

Another approach that can be used with a three-address instruction format is to perform a comparison and specify a branch in the same instruction. For example,

BRE R1, R2, X Branch to X if contents of R1 = contents of R2.

Figure 12.7 shows examples of these operations. Note that a branch can be either *forward* (an instruction with a higher address) or *backward* (lower address). The example shows how an unconditional and a conditional branch can be used to create a repeating loop of instructions. The instructions in locations 202 through 210 will be executed repeatedly until the result of subtracting Y from X is 0.

SKIP INSTRUCTIONS Another form of transfer-of-control instruction is the skip instruction. The skip instruction includes an implied address. Typically, the skip implies that one instruction be skipped; thus, the implied address equals the address of the next instruction plus one instruction length. Because the skip instruction does not require a destination address field, it is free to do other things. A typical example is the increment-and-skip-if-zero (ISZ) instruction. Consider the following program fragment:

```
301
 .
 .
 .
309  ISZ  R1
310  BR   301
311
```

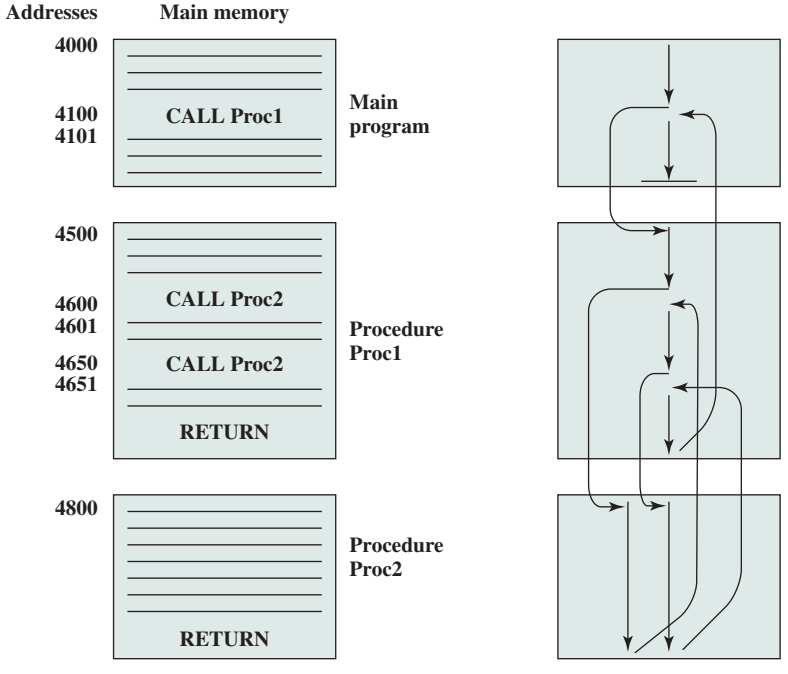
In this fragment, the two transfer-of-control instructions are used to implement an iterative loop. R1 is set with the negative of the number of iterations to be performed. At the end of the loop, R1 is incremented. If it is not 0, the program branches back to the beginning of the loop. Otherwise, the branch is skipped, and the program continues with the next instruction after the end of the loop.

PROCEDURE CALL INSTRUCTIONS Perhaps the most important innovation in the development of programming languages is the *procedure.* A procedure is a self-contained computer program that is incorporated into a larger program. At any point in the program the procedure may be invoked, or *called.* The processor is instructed to go and execute the entire procedure and then return to the point from which the call took place.

The two principal reasons for the use of procedures are economy and modularity. A procedure allows the same piece of code to be used many times. This is important for economy in programming effort and for making the most efficient use of storage space in the system (the program must be stored). Procedures also allow large programming tasks to be subdivided into smaller units. This use of *modularity* greatly eases the programming task.

The procedure mechanism involves two basic instructions: a call instruction that branches from the present location to the procedure, and a return instruction that returns from the procedure to the place from which it was called. Both of these are forms of branching instructions.

Figure 12.8a illustrates the use of procedures to construct a program. In this example, there is a main program starting at location 4000. This program includes a call to procedure PROC1, starting at location 4500. When this call instruction is encountered, the processor suspends execution of the main program and begins execution of PROC1 by fetching the next instruction from location 4500. Within PROC1, there are two calls to PROC2 at location 4800. In each case, the execution of PROC1



(a) Calls and returns

(b) Execution sequence

Figure 12.8 Nested Procedures

is suspended and PROC2 is executed. The RETURN statement causes the processor to go back to the calling program and continue execution at the instruction after the corresponding CALL instruction. This behavior is illustrated in Figure 12.8b.

Three points are worth noting:

1. A procedure can be called from more than one location.
2. A procedure call can appear in a procedure. This allows the *nesting* of procedures to an arbitrary depth.
3. Each procedure call is matched by a return in the called program.

Because we would like to be able to call a procedure from a variety of points, the processor must somehow save the return address so that the return can take place appropriately. There are three common places for storing the return address:

- Register
- Start of called procedure
- Top of stack

Consider a machine-language instruction CALL X, which stands for *call procedure at location X*. If the register approach is used, CALL X causes the following actions:

$$RN \leftarrow PC + \Delta$$
$$PC \leftarrow X$$

where RN is a register that is always used for this purpose, PC is the program counter, and Δ is the instruction length. The called procedure can now save the contents of RN to be used for the later return.

A second possibility is to store the return address at the start of the procedure. In this case, CALL X causes

$$X \leftarrow PC + \Delta$$
$$PC \leftarrow X + 1$$

This is quite handy. The return address has been stored safely away.

Both of the preceding approaches work and have been used. The only limitation of these approaches is that they complicate the use of *reentrant* procedures. A **reentrant procedure** is one in which it is possible to have several calls open to it at the same time. A recursive procedure (one that calls itself) is an example of the use of this feature (see Appendix M). If parameters are passed via registers or memory for a reentrant procedure, some code must be responsible for saving the parameters so that the registers or memory space are available for other procedure calls.

A more general and powerful approach is to use a stack (see Appendix I for a discussion of stacks). When the processor executes a call, it places the return address on the stack. When it executes a return, it uses the address on the stack. Figure 12.9 illustrates the use of the stack.

In addition to providing a return address, it is also often necessary to pass parameters with a procedure call. These can be passed in registers. Another possibility is to store the parameters in memory just after the CALL instruction. In this case, the return must be to the location following the parameters. Again, both of

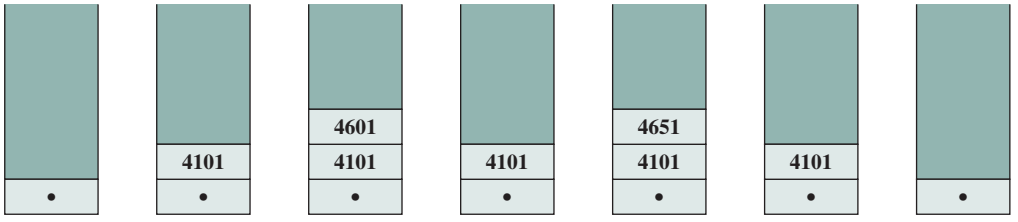


Figure 12.9 Use of Stack to Implement Nested Subroutines of Figure 12.8

these approaches have drawbacks. If registers are used, the called program and the calling program must be written to assure that the registers are used properly. The storing of parameters in memory makes it difficult to exchange a variable number of parameters. Both approaches prevent the use of reentrant procedures.

A more flexible approach to parameter passing is the stack. When the processor executes a call, it not only stacks the return address, it stacks parameters to be passed to the called procedure. The called procedure can access the parameters from the stack. Upon return, return parameters can also be placed on the stack. The entire set of parameters, including return address, that is stored for a procedure invocation is referred to as a *stack frame.*

An example is provided in Figure 12.10. The example refers to procedure P in which the local variables $x1$ and $x2$ are declared, and procedure Q, which P can call and in which the local variables $y1$ and $y2$ are declared. In this figure, the return

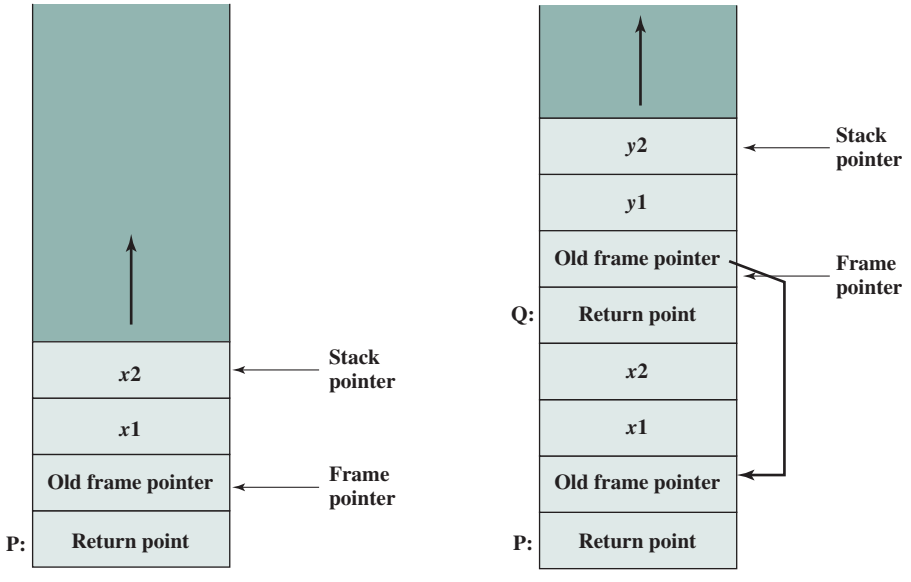


Figure 12.10 Stack Frame Growth Using Sample Procedures P and Q

point for each procedure is the first item stored in the corresponding stack frame. Next is stored a pointer to the beginning of the previous frame. This is needed if the number or length of parameters to be stacked is variable.

12.5 INTEL x86 AND ARM OPERATION TYPES

x86 Operation Types

The x86 provides a complex array of operation types, including a number of specialized instructions. The intent was to provide tools for the compiler writer to produce optimized machine language translation of high-level language programs. Most of these are the conventional instructions found in most machine instruction sets, but several types of instructions are tailored to the x86 architecture and are of particular interest. Appendix A of [CART06] lists the x86 instructions, together with the operands for each and the effect of the instruction on the condition codes. Appendix B of the NASM assembly language manual [NASM12] provides a more detailed description of each x86 instruction. Both documents are available at box.com/COA10e.

CALL/RETURN INSTRUCTIONS The x86 provides four instructions to support procedure call/return: CALL, ENTER, LEAVE, RETURN. It will be instructive to look at the support provided by these instructions. Recall from Figure 12.10 that a common means of implementing the procedure call/return mechanism is via the use of stack frames. When a new procedure is called, the following must be performed upon entry to the new procedure:

- Push the return point on the stack.
- Push the current frame pointer on the stack.
- Copy the stack pointer as the new value of the frame pointer.
- Adjust the stack pointer to allocate a frame.

The CALL instruction pushes the current instruction pointer value onto the stack and causes a jump to the entry point of the procedure by placing the address of the entry point in the instruction pointer. In the 8088 and 8086 machines, the typical procedure began with the sequence

```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, space_for_locals
```

where EBP is the frame pointer and ESP is the stack pointer. In the 80286 and later machines, the ENTER instruction performs all the aforementioned operations in a single instruction.

The ENTER instruction was added to the instruction set to provide direct support for the compiler. The instruction also includes a feature for support of what are called nested procedures in languages such as Pascal, COBOL, and Ada (not found in C or FORTRAN). It turns out that there are better ways of handling nested procedure calls for these languages. Furthermore, although the ENTER instruction

saves a few bytes of memory compared with the PUSH, MOV, SUB sequence (4 bytes versus 6 bytes), it actually takes longer to execute (10 clock cycles versus 6 clock cycles). Thus, although it may have seemed a good idea to the instruction set designers to add this feature, it complicates the implementation of the processor while providing little or no benefit. We will see that, in contrast, a RISC approach to processor design would avoid complex instructions such as ENTER and might produce a more efficient implementation with a sequence of simpler instructions.

MEMORY MANAGEMENT Another set of specialized instructions deals with memory segmentation. These are privileged instructions that can only be executed from the operating system. They allow local and global segment tables (called descriptor tables) to be loaded and read, and for the privilege level of a segment to be checked and altered.

The special instructions for dealing with the on-chip cache were discussed in Chapter 4.

STATUS FLAGS AND CONDITION CODES Status flags are bits in special registers that may be set by certain operations and used in conditional branch instructions. The term *condition code* refers to the settings of one or more status flags. In the x86 and many other architectures, status flags are set by arithmetic and compare operations. The compare operation in most languages subtracts two operands, as does a subtract operation. The difference is that a compare operation only sets status flags, whereas a subtract operation also stores the result of the subtraction in the destination operand. Some architectures also set status flags for data transfer instructions.

Table 12.8 lists the status flags used on the x86. Each flag, or combinations of these flags, can be tested for a conditional jump. Table 12.9 shows the condition codes (combinations of status flag values) for which conditional jump opcodes have been defined.

Several interesting observations can be made about this list. First, we may wish to test two operands to determine if one number is bigger than another. But this will depend on whether the numbers are signed or unsigned. For example, the 8-bit number 11111111 is bigger than 00000000 if the two numbers are interpreted

Table 12.8 x86 Status Flags

Status Bit	Name	Description
C	Carry	Indicates carrying or borrowing out of the left-most bit position following an arithmetic operation. Also modified by some of the shift and rotate operations.
P	Parity	Parity of the least-significant byte of the result of an arithmetic or logic operation. 1 indicates even parity; 0 indicates odd parity.
A	Auxiliary Carry	Represents carrying or borrowing between half-bytes of an 8-bit arithmetic or logic operation. Used in binary-coded decimal arithmetic.
Z	Zero	Indicates that the result of an arithmetic or logic operation is 0.
S	Sign	Indicates the sign of the result of an arithmetic or logic operation.
O	Overflow	Indicates an arithmetic overflow after an addition or subtraction for twos complement arithmetic.

Table 12.9 x86 Condition Codes for Conditional Jump and SETcc Instructions

Symbol	Condition Tested	Comment
A, NBE	C = 0 AND Z = 0	Above; Not below or equal (greater than, unsigned)
AE, NB, NC	C = 0	Above or equal; Not below (greater than or equal, unsigned); Not carry
B, NAE, C	C = 1	Below; Not above or equal (less than, unsigned); Carry set
BE, NA	C = 1 OR Z = 1	Below or equal; Not above (less than or equal, unsigned)
E, Z	Z = 1	Equal; Zero (signed or unsigned)
G, NLE	[(S = 1 AND O = 1) OR (S = 0 AND O = 0)]AND[Z = 0]	Greater than; Not less than or equal (signed)
GE, NL	(S = 1 AND O = 1) OR (S = 0 AND O = 0)	Greater than or equal; Not less than (signed)
L, NGE	(S = 1 AND O = 0) OR (S = 0 AND O = 0)	Less than; Not greater than or equal (signed)
LE, NG	(S = 1 AND O = 0) OR (S = 0 AND O = 1) OR (Z = 1)	Less than or equal; Not greater than (signed)
NE, NZ	Z = 0	Not equal; Not zero (signed or unsigned)
NO	O = 0	No overflow
NS	S = 0	Not sign (not negative)
NP, PO	P = 0	Not parity; Parity odd
O	O = 1	Overflow
P	P = 1	Parity; Parity even
S	S = 1	Sign (negative)

as unsigned integers ($255 > 0$) but is less if they are considered as 8-bit twos complement numbers ($-1 < 0$). Many assembly languages therefore introduce two sets of terms to distinguish the two cases: If we are comparing two numbers as signed integers, we use the terms *less than* and *greater than;* if we are comparing them as unsigned integers, we use the terms *below* and *above.*

A second observation concerns the complexity of comparing signed integers. A signed result is greater than or equal to zero if (1) the sign bit is zero and there is no overflow (S = 0 AND O = 0), or (2) the sign bit is one and there is an overflow. A study of Figure 10.4 should convince you that the conditions tested for the various signed operations are appropriate.

x86 SIMD INSTRUCTIONS In 1996, Intel introduced MMX technology into its Pentium product line. MMX is set of highly optimized instructions for multimedia tasks. There are 57 new instructions that treat data in a SIMD (single-instruction, multiple-data) fashion, which makes it possible to perform the same operation, such as addition or multiplication, on multiple data elements at once. Each instruction typically takes a single clock cycle to execute. For the proper application, these fast parallel operations can yield a speedup of two to eight times over comparable algorithms that do not use the MMX instructions [ATKI96]. With the introduction of 64-bit x86 architecture, Intel has expanded this extension to include double

quadword (128 bits) operands and floating-point operations. In this subsection, we describe the MMX features.

The focus of MMX is multimedia programming. Video and audio data are typically composed of large arrays of small data types, such as 8 or 16 bits, whereas conventional instructions are tailored to operate on 32- or 64-bit data. Here are some examples: In graphics and video, a single scene consists of an array of pixels,[2] and there are 8 bits for each pixel or 8 bits for each pixel color component (red, green, blue). Typical audio samples are quantized using 16 bits. For some 3D graphics algorithms, 32 bits are common for basic data types. To provide for parallel operation on these data lengths, three new data types are defined in MMX. Each data type is 64 bits in length and consists of multiple smaller data fields, each of which holds a fixed-point integer. The types are as follows:

- **Packed byte:** Eight bytes packed into one 64-bit quantity.
- **Packed word:** Four 16-bit words packed into 64 bits.
- **Packed doubleword:** Two 32-bit doublewords packed into 64 bits.

Table 12.10 lists the MMX instruction set. Most of the instructions involve parallel operation on bytes, words, or doublewords. For example, the PSLLW instruction performs a left logical shift separately on each of the four words in the packed word operand; the PADDB instruction takes packed byte operands as input and performs parallel additions on each byte position independently to produce a packed byte output.

One unusual feature of the new instruction set is the introduction of **saturation arithmetic** for byte and 16-bit word operands. With ordinary unsigned arithmetic, when an operation overflows (i.e., a carry out of the most significant bit), the extra bit is truncated. This is referred to as wraparound, because the effect of the truncation can be, for example, to produce an addition result that is smaller than the two input operands. Consider the addition of the two words, in hexadecimal, F000h and 3000h. The sum would be expressed as

$$\begin{aligned} \text{F000h} &= 1111\ 0000\ 0000\ 0000 \\ +\ \text{3000h} &= \underline{0011\ 0000\ 0000\ 0000} \\ & \quad 10010\ 0000\ 0000\ 0000 = \text{2000h} \end{aligned}$$

If the two numbers represented image intensity, then the result of the addition is to make the combination of two dark shades turn out to be lighter. This is typically not what is intended. With saturation arithmetic, if addition results in overflow or subtraction results in underflow, the result is set to the largest or smallest value representable. For the preceding example, with saturation arithmetic, we have

$$\begin{aligned} \text{F000h} &= 1111\ 0000\ 0000\ 0000 \\ +\ \text{3000h} &= \underline{0011\ 0000\ 0000\ 0000} \\ & \quad 10010\ 0000\ 0000\ 0000 \\ & \quad 1111\ 1111\ 1111\ 1111 = \text{FFFFh} \end{aligned}$$

[2]A pixel, or picture element, is the smallest element of a digital image that can be assigned a gray level. Equivalently, a pixel is an individual dot in a dot-matrix representation of a picture.

Table 12.10 MMX Instruction Set

Category	Instruction	Description
Arithmetic	PADD [B, W, D]	Parallel add of packed eight bytes, four 16-bit words, or two 32-bit doublewords, with wraparound.
	PADDS [B, W]	Add with saturation.
	PADDUS [B, W]	Add unsigned with saturation.
	PSUB [B, W, D]	Subtract with wraparound.
	PSUBS [B, W]	Subtract with saturation.
	PSUBUS [B, W]	Subtract unsigned with saturation.
	PMULHW	Parallel multiply of four signed 16-bit words, with high-order 16 bits of 32-bit result chosen.
	PMULLW	Parallel multiply of four signed 16-bit words, with low-order 16 bits of 32-bit result chosen.
	PMADDWD	Parallel multiply of four signed 16-bit words; add together adjacent pairs of 32-bit results.
Comparison	PCMPEQ [B, W, D]	Parallel compare for equality; result is mask of 1s if true or 0s if false.
	PCMPGT [B, W, D]	Parallel compare for greater than; result is mask of 1s if true or 0s if false.
Conversion	PACKUSWB	Pack words into bytes with unsigned saturation.
	PACKSS [WB, DW]	Pack words into bytes, or doublewords into words, with signed saturation.
	PUNPCKH [BW, WD, DQ]	Parallel unpack (interleaved merge) high-order bytes, words, or doublewords from MMX register.
	PUNPCKL [BW, WD, DQ]	Parallel unpack (interleaved merge) low-order bytes, words, or doublewords from MMX register.
Logical	PAND	64-bit bitwise logical AND
	PNDN	64-bit bitwise logical AND NOT
	POR	64-bit bitwise logical OR
	PXOR	64-bit bitwise logical XOR
Shift	PSLL [W, D, Q]	Parallel logical left shift of packed words, doublewords, or quadword by amount specified in MMX register or immediate value.
	PSRL [W, D, Q]	Parallel logical right shift of packed words, doublewords, or quadword.
	PSRA [W, D]	Parallel arithmetic right shift of packed words, doublewords, or quadword.
Data transfer	MOV [D, Q]	Move doubleword or quadword to/from MMX register.
Statemgt	EMMS	Empty MMX state (empty FP registers tag bits).

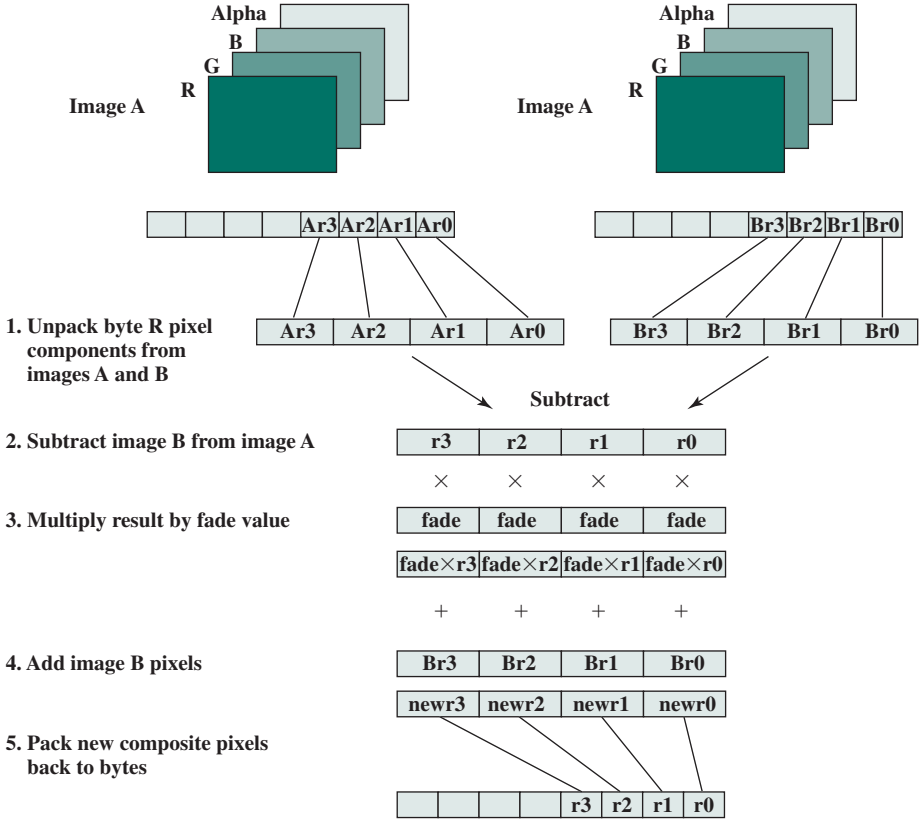
Note: If an instruction supports multiple data types [byte (B), word (W), doubleword (D), quadword (Q)], the data types are indicated in brackets.

To provide a feel for the use of MMX instructions, we look at an example, taken from [PELE97]. A common video application is the fade-out, fade-in effect, in which one scene gradually dissolves into another. Two images are combined with a weighted average:

$$\text{Result\_pixel} = \text{A\_pixel} \times \text{fade} + \text{B\_pixel} \times (1 - \text{fade})$$

This calculation is performed on each pixel position in A and B. If a series of video frames is produced while gradually changing the fade value from 1 to 0 (scaled appropriately for an 8-bit integer), the result is to fade from image A to image B.

Figure 12.11 shows the sequence of steps required for one set of pixels. The 8-bit pixel components are converted to 16-bit elements to accommodate the MMX 16-bit multiply capability. If these images use 640 × 480 resolution, and the dissolve technique uses all 255 possible values of the fade value, then the total number of



MMX code sequence performing this operation:

```
pxor      mm7, mm7     ;zero out mm7
movq      mm3, fad_val ;load fade value replicated 4 times
movd      mm0, imageA  ;load 4 red pixel components from image A
movd      mm1, imageB  ;load 4 red pixel components from image B
punpckblw mm0, mm7     ;unpack 4 pixels to 16 bits
punpckblw mm1, mm7     ;unpack 4 pixels to 16 bits
psubw     mm0, mm1     ;subtract image B from image A
pmulhw    mm0, mm3     ;multiply the subtract result by fade values
padddw    mm0, mm1     ;add result to image B
packuswb  mm0, mm7     ;pack 16-bit results back to bytes
```

Figure 12.11 Image Compositing on Color Plane Representation

instructions executed using MMX is 535 million. The same calculation, performed without the MMX instructions, requires 1.4 billion instruction executions [INTE98].

ARM Operation Types

The ARM architecture provides a large collection of operation types. The following are the principal categories:

- **Load and store instructions:** In the ARM architecture, only load and store instructions access memory locations; arithmetic and logical instructions are performed only on registers and immediate values encoded in the instruction. This limitation is characteristic of RISC design and it is explored further in Chapter 15. The ARM architecture supports two broad types of instruction that load or store the value of a single register, or a pair of registers, from or to memory: (1) load or store a 32-bit word or an 8-bit unsigned byte, and (2) load or store a 16-bit unsigned halfword, and load and sign extend a 16-bit halfword or an 8-bit byte.
- **Branch instructions:** ARM supports a branch instruction that allows a conditional branch forwards or backwards up to 32 MB. A subroutine call can be performed by a variant of the standard branch instruction. As well as allowing a branch forward or backward up to 32 MB, the Branch with Link (BL) instruction preserves the address of the instruction after the branch (the return address) in the LR (R14). Branches are determined by a 4-bit condition field in the instruction.
- **Data-processing instructions:** This category includes logical instructions (AND, OR, XOR), add and subtract instructions, and test and compare instructions.
- **Multiply instructions:** The integer multiply instructions operate on word or halfword operands and can produce normal or long results. For example, there is a multiply instruction that takes two 32-bit operands and produces a 64-bit result.
- **Parallel addition and subtraction instructions:** In addition to the normal data processing and multiply instructions, there are a set of parallel addition and subtraction instructions, in which portions of two operands are operated on in parallel. For example, ADD16 adds the top halfwords of two registers to form the top halfword of the result and adds the bottom halfwords of the same two registers to form the bottom halfword of the result. These instructions are useful in image processing applications, similar to the x86 MMX instructions.
- **Extend instructions:** There are several instructions for unpacking data by sign or zero extending bytes to halfwords or words, and halfwords to words.
- **Status register access instructions:** ARM provides the ability to read and also to write portions of the status register.

CONDITION CODES The ARM architecture defines four condition flags that are stored in the program status register: N, Z, C, and V (Negative, Zero, Carry and oVerflow), with meanings essentially the same as the S, Z, C, and V flags in the

Table 12.11 ARM Conditions for Conditional Instruction Execution

Code	Symbol	Condition Tested	Comment
0000	EQ	Z = 1	Equal
0001	NE	Z = 0	Not equal
0010	CS/HS	C = 1	Carry set/unsigned higher or same
0011	CC/LO	C = 0	Carry clear/unsigned lower
0100	MI	N = 1	Minus/negative
0101	PL	N = 0	Plus/positive or zero
0110	VS	V = 1	Overflow
0111	VC	V = 0	No overflow
1000	HI	C = 1 AND Z = 0	Unsigned higher
1001	LS	C = 0 OR Z = 1	Unsigned lower or same
1010	GE	N = V [(N = 1 AND V = 1) OR (N = 0 AND V = 0)]	Signed greater than or equal
1011	LT	N ≠ V [(N = 1 AND V = 0) OR (N = 0 AND V = 1)]	Signed less than
1100	GT	(Z = 0) AND (N = V)	Signed greater than
1101	LE	(Z = 1) OR (N ≠ V)	Signed less than or equal
1110	AL	—	Always (unconditional)
1111	—	—	This instruction can only be executed unconditionally

x86 architecture. These four flags constitute a condition code in ARM. Table 12.11 shows the combination of conditions for which conditional execution is defined.

There are two unusual aspects to the use of condition codes in ARM:

1. All instructions, not just branch instructions, include a condition code field, which means that virtually all instructions may be conditionally executed. Any combination of flag settings except 1110 or 1111 in an instruction's condition code field signifies that the instruction will be executed only if the condition is met.
2. All data processing instructions (arithmetic, logical) include an S bit that signifies whether the instruction updates the condition flags.

The use of conditional execution and conditional setting of the condition flags helps in the design of shorter programs that use less memory. On the other hand, all instructions include 4 bits for the condition code, so there is a trade-off in that fewer bits in the 32-bit instruction are available for opcode and operands. Because the ARM is a RISC design that relies heavily on register addressing, this seems to be a reasonable trade-off.

12.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

accumulator	jump	procedure call
address	little endian	procedure return
arithmetic shift	logical shift	push
bi-endian	machine instruction	reentrant procedure
big endian	operand	rotate
branch	operation	skip
conditional branch	packed decimal	stack
instruction set	pop	

Review Questions

12.1 What are the typical elements of a machine instruction?
12.2 What types of locations can hold source and destination operands?
12.3 If an instruction contains four addresses, what might be the purpose of each address?
12.4 List and briefly explain five important instruction set design issues.
12.5 What types of operands are typical in machine instruction sets?
12.6 What is the relationship between the IRA character code and the packed decimal representation?
12.7 What is the difference between an arithmetic shift and a logical shift?
12.8 Why are transfer of control instructions needed?
12.9 List and briefly explain two common ways of generating the condition to be tested in a conditional branch instruction.
12.10 What is meant by the term *nesting of procedures*?
12.11 List three possible places for storing the return address for a **procedure return**.
12.12 What is a reentrant procedure?
12.13 What is **reverse Polish notation**?
12.14 What is the difference between big endian and little endian?

Problems

12.1 Show in hex notation:
- **a.** The packed decimal format for 23.
- **b.** The ASCII characters 23.

12.2 For each of the following packed decimal numbers, show the decimal value:
- **a.** 0111 0011 0000 1001
- **b.** 0101 1000 0010
- **c.** 0100 1010 0110

12.3 A given microprocessor has words of 1 byte. What is the smallest and largest integer that can be represented in the following representations:
- **a.** Unsigned.
- **b.** Sign-magnitude.
- **c.** Ones complement.
- **d.** Twos complement.

e. Unsigned packed decimal.
f. Signed packed decimal.

12.4 Many processors provide logic for performing arithmetic on packed decimal numbers. Although the rules for decimal arithmetic are similar to those for binary operations, the decimal results may require some corrections to the individual digits if binary logic is used.

Consider the decimal addition of two unsigned numbers. If each number consists of N digits, then there are $4N$ bits in each number. The two numbers are to be added using a binary adder. Suggest a simple rule for correcting the result. Perform addition in this fashion on the numbers 1698 and 1786.

12.5 The tens complement of the decimal number X is defined to be $10^N - X$, where N is the number of decimal digits in the number. Describe the use of ten's complement representation to perform decimal subtraction. Illustrate the procedure by subtracting $(0326)_{10}$ from $(0736)_{10}$.

12.6 Compare zero-, one-, two-, and three-address machines by writing programs to compute

$$X = (A + B \times C)/(D - E \times F)$$

for each of the four machines. The instructions available for use are as follows:

0 Address	1 Address	2 Address	3 Address
PUSH M	LOAD M	MOVE (X ← Y)	MOVE (X ← Y)
POP M	STORE M	ADD (X ← X + Y)	ADD (X ← Y + Z)
ADD	ADD M	SUB (X ← X − Y)	SUB (X ← Y − Z)
SUB	SUB M	MUL (X ← X × Y)	MUL (X ← Y × Z)
MUL	MUL M	DIV (X ← X/Y)	DIV (X ← Y/Z)
DIV	DIV M		

12.7 Consider a hypothetical computer with an instruction set of only two n-bit instructions. The first bit specifies the opcode, and the remaining bits specify one of the 2^{n-1} n-bit words of main memory. The two instructions are as follows:

SUBS X Subtract the contents of location X from the accumulator, and store the result in location X and the accumulator.

JUMP X Place address X in the program counter.

A word in main memory may contain either an instruction or a binary number in twos complement notation. Demonstrate that this instruction repertoire is reasonably complete by specifying how the following operations can be programmed:
a. Data transfer: Location X to accumulator, accumulator to location X.
b. Addition: Add contents of location X to accumulator.
c. Conditional branch.
d. Logical OR.
e. I/O Operations.

12.8 Many instruction sets contain the instruction NOOP, meaning no operation, which has no effect on the processor state other than incrementing the program counter. Suggest some uses of this instruction.

12.9 In Section 12.4, it was stated that both an arithmetic left shift and a logical left shift correspond to a multiplication by 2 when there is no overflow, and if overflow occurs, arithmetic and logical left shift operations produce different results, but the arithmetic left shift retains the sign of the number. Demonstrate that these statements are true for 5-bit twos complement integers.

12.10 In what way are numbers rounded using arithmetic right shift (e.g., round toward $+\infty$, round toward $-\infty$, toward zero, away from 0)?

12.11 Suppose a stack is to be used by the processor to manage procedure calls and returns. Can the program counter be eliminated by using the top of the stack as a program counter?

12.12 The x86 architecture includes an instruction called Decimal Adjust after Addition (DAA). DAA performs the following sequence of instructions:

```
if ((AL AND 0FH) >9) OR (AF = 1)  then
      AL ← AL + 6;
      AF ← 1;
else
      AF ← 0;
endif;
if (AL > 9FH) OR (CF = 1)  then
      AL ← AL + 60H;
      CF ← 1;
else
      CF ← 0;
endif.
```

"H" indicates hexadecimal. AL is an 8-bit register that holds the result of addition of two unsigned 8-bit integers. AF is a flag set if there is a carry from bit 3 to bit 4 in the result of an addition. CF is a flag set if there is a carry from bit 7 to bit 8. Explain the function performed by the DAA instruction.

12.13 The x86 Compare instruction (CMP) subtracts the source operand from the destination operand; it updates the status flags (C, P, A, Z, S, O) but does not alter either of the operands. The CMP instruction can be used to determine if the destination operand is greater than, equal to, or less than the source operand.

a. Suppose the two operands are treated as unsigned integers. Show which status flags are relevant to determine the relative size of the two integer and what values of the flags correspond to greater than, equal to, or less than.

b. Suppose the two operands are treated as twos complement signed integers. Show which status flags are relevant to determine the relative size of the two integer and what values of the flags correspond to greater than, equal to, or less than.

c. The CMP instruction may be followed by a conditional Jump (Jcc) or Set Condition (SETcc) instruction, where cc refers to one of the 16 conditions listed in Table 12.11. Demonstrate that the conditions tested for a signed number comparison are correct.

12.14 Suppose we wished to apply the x86 CMP instruction to 32-bit operands that contained numbers in a floating-point format. For correct results, what requirements have to be met in the following areas?

a. The relative position of the significand, sign, and exponent fields.

b. The representation of the value zero.

c. The representation of the exponent.

d. Does the IEEE format meet these requirements? Explain.

12.15 Many microprocessor instruction sets include an instruction that tests a condition and sets a destination operand if the condition is true. Examples include the SETcc on the x86, the Scc on the Motorola MC68000, and the Scond on the National NS32000.

a. There are a few differences among these instructions:

- SETcc and Scc operate only on a byte, whereas Scond operates on byte, word, and doubleword operands.
- SETcc and Scond set the operand to integer one if true and to zero if false. Scc sets the byte to all binary ones if true and all zeros if false. What are the relative advantages and disadvantages of these differences?

b. None of these instructions set any of the condition code flags, and thus an explicit test of the result of the instruction is required to determine its value. Discuss whether condition codes should be set as a result of this instruction.

c. A simple IF statement such as IF a > b THEN can be implemented using a numerical representation method, that is, making the Boolean value manifest, as opposed to a *flow of control* method, which represents the value of a Boolean expression by a point reached in the program. A compiler might implement IF a > ssb THEN with the following x86 code:

```
        SUB   CX, CX    ;set register CX to 0
        MOV   AX, B     ;move contents of location B to register AX
        CMP   AX, A     ;compare contents of register AX and location A
        JLE   TEST      ;jump if A ≤ B
        INC   CX        ;add 1 to contents of register CX
TEST    JCXZ  OUT       ;jump if contents of CX equal 0
THEN          OUT
```

The result of (A > B) is a Boolean value held in a register and available later on, outside the context of the flow of code just shown. It is convenient to use register CX for this, because many of the branch and loop opcodes have a built-in test for CX.

Show an alternative implementation using the SETcc instruction that saves memory and execution time. (*Hint:* No additional new x86 instructions are needed, other than the SETcc.)

d. Now consider the high-level language statement:

$$A := (B > C) \text{ OR } (D = F)$$

A compiler might generate the following code:

```
        MOV   EAX, B    ;move contents of location B to register EAX
        CMP   EAX, C    ;compare contents of register EAX and location C
        MOV   BL, 0     ;0 represents false
        JLE   N1        ;jump if (B ≤ C)
        MOV   BL, 1     ;1 represents false
N1      MOV   EAX, D
        CMP   EAX, F
        MOV   BH, 0
        JNE   N2
        MOV   BH, 1
N2      OR    BL, BH
```

Show an alternative implementation using the SETcc instruction that saves memory and execution time.

12.16 Suppose that two registers contain the following hexadecimal values: AB0890C2, 4598EE50. What is the result of adding them using MMX instructions:

a. packed byte.

b. packed word.

Assume saturation arithmetic is not used.

12.17 Appendix I points out that there are no stack-oriented instructions in an instruction set if the stack is to be used only by the processor for such purposes as procedure handling. How can the processor use a stack for any purpose without stack-oriented instructions?

12.18 Mathematical formulas are usually expressed in what is known as infix notation, in which a binary operator appears between the operands. An alternative technique is known as **reverse Polish**, or **postfix**, notation, in which the operator follows its two operands. See Appendix I for more details. Convert the following formulas from reverse Polish to infix:

a. AB + C + D ×
b. AB/CD/ +
c. ABCDE + × × /
d. ABCDE + F/ + G − H/ × +

12.19 Convert the following formulas from infix to reverse Polish:

a. A + B + C + D + E
b. (A + B) × (C + D) + E
c. (A × B) + (C × D) + E
d. (A − B) × (((C − D × E)/F)/G) × H

12.20 Convert the expression A + B − C to postfix notation using Dijkstra's algorithm. Show the steps involved. Is the result equivalent to (A + B) − C or A + (B − C)? Does it matter?

12.21 Using the algorithm for converting infix to postfix defined in Appendix I, show the steps involved in converting the expression of Figure I.3 into postfix. Use a presentation similar to Figure I.5.

12.22 Show the calculation of the expression in Figure I.5, using a presentation similar to Figure I.4.

12.23 Redraw the little-endian layout in Figure 12.13 so that the bytes appear as numbered in the big-endian layout. That is, show memory in 64-bit rows, with the bytes listed left to right, top to bottom.

12.24 For the following data structures, draw the big-endian and little-endian layouts, using the format of Figure 12.13, and comment on the results.

a.
```
struct {
   double i;     //0x1112131415161718
} s1;
```

b.
```
struct {
   int i;        //0x11121314
   int j;        //0x15161718
} s2;
```

c.
```
struct {
   short i;      //0x1112
   short j;      //0x1314
   short k;      //0x1516
   short l;      //0x1718
} s3;
```

12.25 The IBM Power architecture specification does not dictate how a processor should implement little-endian mode. It specifies only the view of memory a processor must have when operating in little-endian mode. When converting a data structure from big endian to little endian, processors are free to implement a true byte-swapping mechanism or to use some sort of an address modification mechanism. Current Power processors are all default big-endian machines and use address modification to treat data as little-endian.

Consider the structure s defined in Figure 12.13. The layout in the lower-right portion of the figure shows the structure s as seen by the processor. In fact, if structures is compiled in little-endian mode, its layout in memory is shown in Figure 12.12.

Little-endian address mapping

Byte address								
					11	12	13	14
00	00	01	02	03	04	05	06	07
	21	22	23	24	25	26	27	28
08	08	09	0A	0B	0C	0D	0E	0F
	'D'	'C'	'B'	'A'	31	32	33	34
10	10	11	12	13	14	15	16	17
			51	52		'G'	'F'	'E'
18	18	19	1A	1B	1C	1D	1E	1F
					61	62	63	64
20	20	21	22	23	24	25	26	27

Figure 12.12 Power Architecture Little-Endian Structures in Memory

Explain the mapping that is involved, describe an easy way to implement the mapping, and discuss the effectiveness of this approach.

12.26 Write a small program to determine the endianness of machine and report the results. Run the program on a computer available to you and turn in the output.

12.27 The MIPS processor can be set to operate in either big-endian or little-endian mode. Consider the Load Byte Unsigned (LBU) instruction, which loads a byte from memory into the low-order 8 bits of a register and fills the high-order 24 bits of the register with zeros. The description of LBU is given in the MIPS reference manual using a register-transfer language as

```
mem ← LoadMemory(…)
byte ← VirtualAddress1..0
if  CONDITION  then
       GPR[rt] ← 0^24 || mem31 - 8 × byte .. 24 - 8 × byte
else
       GPR[rt] ← 0^24 || mem7 + 8 × byte .. 8 × byte
endif
```

where *byte* refers to the two low-order bits of the effective address and *mem* refers to the value loaded from memory. In the manual, instead of the word CONDITION, one of the following two words is used: BigEndian, LittleEndian. Which word is used?

12.28 Most, but not all, processors use big- or little-endian bit ordering within a byte that is consistent with big- or little-endian ordering of bytes within a multibyte scalar. Let us consider the Motorola 68030, which uses big-endian byte ordering. The documentation of the 68030 concerning formats is confusing. The user's manual explains that the bit ordering of bit fields is the opposite of bit ordering of integers. Most bit field operations operate with one endian ordering, but a few bit field operations require the opposite ordering. The following description from the user's manual describes most of the bit field operations:

> A bit operand is specified by a base address that selects one byte in memory (the base byte), and a bit number that selects the one bit in this byte. The most significant bit is bit seven. A bit field operand is specified by: **(1)** a base address that selects one byte in memory; **(2)** a bit field offset that indicates the leftmost (base) bit of the bit field in relation to the most significant bit of the base byte; and **(3)** a bit field width that determines how many bits to the right of the base byte are in the bit field. The most significant bit of the base byte is bit field offset 0, the least significant bit of the base byte is bit field offset 7.

Do these instructions use big-endian or little-endian bit ordering?

APENDIX 12A LITTLE-, BIG-, AND BI-ENDIAN

An annoying and curious phenomenon relates to how the bytes within a word and the bits within a byte are both referenced and represented. We look first at the problem of byte ordering and then consider that of bits.

Byte Ordering

The concept of endianness was first discussed in the literature by Cohen [COHE81]. With respect to bytes, endianness has to do with the byte ordering of multibyte scalar values. The issue is best introduced with an example. Suppose we have the 32-bit hexadecimal value 12345678 and that it is stored in a 32-bit word in byte-addressable memory at byte location 184. The value consists of 4 bytes, with the least significant byte containing the value 78 and the most significant byte containing the value 12. There are two obvious ways to store this value:

Address	Value
184	12
185	34
186	56
187	78

Address	Value
184	78
185	56
186	34
187	12

The mapping on the left stores the most significant byte in the lowest numerical byte address; this is known as **big endian** and is equivalent to the left-to-right order of writing in Western culture languages. The mapping on the right stores the least significant byte in the lowest numerical byte address; this is known as **little endian** and is reminiscent of the right-to-left order of arithmetic operations in arithmetic units.[3] For a given multibyte scalar value, big endian and little endian are byte-reversed mappings of each other.

The concept of endianness arises when it is necessary to treat a multiple-byte entity as a single data item with a single address, even though it is composed of smaller addressable units. Some machines, such as the Intel 80x86, x86, VAX, and Alpha, are little-endian machines, whereas others, such as the IBM System 370/390, the Motorola 680x0, Sun SPARC, and most RISC machines, are big endian. This presents problems when data are transferred from a machine of one endian type to the other and when a programmer attempts to manipulate individual bytes or bits within a multibyte scalar.

The property of endianness does not extend beyond an individual data unit. In any machine, aggregates such as files, data structures, and arrays are composed of multiple data units, each with endianness. Thus, conversion of a block of memory from one style of endianness to the other requires knowledge of the data structure.

Figure 12.13 illustrates how endianness determines addressing and byte order. The C structure at the top contains a number of data types. The memory layout in the

[3]The terms *big endian* and *little endian* come from Part I, Chapter 4 of Jonathan Swift's *Gulliver's Travels*. They refer to a religious war between two groups, one that breaks eggs at the big end and the other that breaks eggs at the little end.

```
struct{
   int     a;     //0x1112_1314                    word
   int     pad;   //
   double  b;     //0x2122_2324_2526_2728          doubleword
   char*   c;     //0x3132_3334                    word
   char    d[7];  //'A','B','C','D','E','F','G'    byte array
   short   e;     //0x5152                         halfword
   int     f;     //0x6162_6364                    word
} s;
```

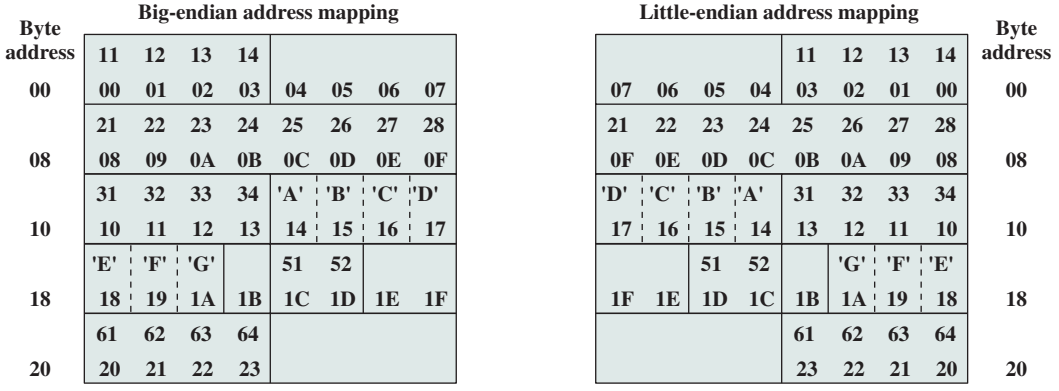


Figure 12.13 Example C Data Structure and Its Endian Maps

lower left results from compilation of that structure for a big-endian machine, and that in the lower right for a little-endian machine. In each case, memory is depicted as a series of 64-bit rows. For the big-endian case, memory typically is viewed left to right, top to bottom, whereas for the little-endian case, memory typically is viewed as right to left, top to bottom. Note that these layouts are arbitrary. Either scheme could use either left to right or right to left within a row; this is a matter of depiction, not memory assignment. In fact, in looking at programmer manuals for a variety of machines, a bewildering collection of depictions is to be found, even within the same manual.

```
struct{
   int a; //0x1112_1314                    word
   int pad; //
   double b; //0x2122_2324_2526_2728       doubleword
   char* c; //0x3132_3334                  word
   char d[7]; //'A','B','C','D','E','F','G'  byte array
   short e; //0x5152                       halfword
   int f; //0x6162_6364                    word
} s;
```

We can make several observations about this data structure:

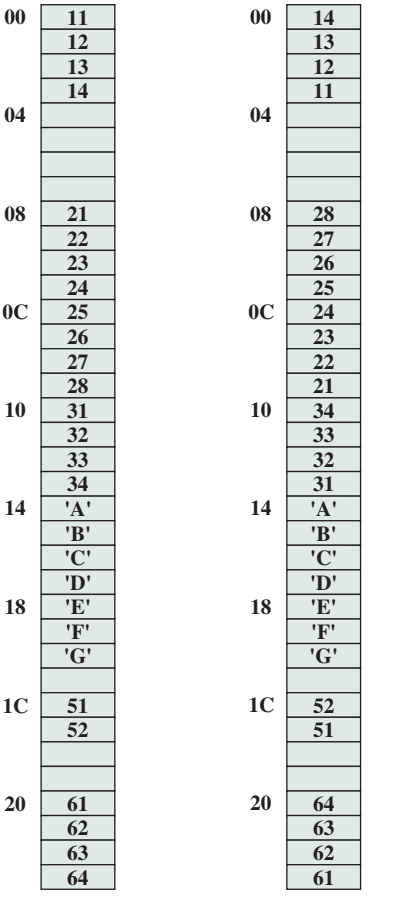
- Each data item has the same address in both schemes. For example, the address of the doubleword with hexadecimal value 2122232425262728 is 08.
- Within any given multibyte scalar value, the ordering of bytes in the little-endian structure is the reverse of that for the big-endian structure.

- Endianness does not affect the ordering of data items within a structure. Thus, the four-character word c exhibits byte reversal, but the seven-character byte array d does not. Hence, the address of each individual element of d is the same in both structures.

The effect of endianness is perhaps more clearly demonstrated when we view memory as a vertical array of bytes, as shown in Figure 12.14.

There is no general consensus as to which is the superior style of endianness.[4] The following points favor the big-endian style:

- **Character-string sorting:** A big-endian processor is faster in comparing integer-aligned character strings; the integer ALU can compare multiple bytes in parallel.



(a) Big endian (b) Little endian

Figure 12.14 Another View of Figure 12.13

[4]The prophet revered by both groups in the Endian Wars of *Gulliver's Travels* had this to say. "All true Believers shall break their Eggs at the convenient End." Not much help!

- **Decimal/IRA dumps:** All values can be printed left to right without causing confusion.
- **Consistent order:** Big-endian processors store their integers and character strings in the same order (most significant byte comes first).

The following points favor the little-endian style:

- A big-endian processor has to perform addition when it converts a 32-bit integer address to a 16-bit integer address, to use the least significant bytes.
- It is easier to perform higher-precision arithmetic with the little-endian style; you don't have to find the least-significant byte and move backward.

The differences are minor and the choice of endian style is often more a matter of accommodating previous machines than anything else.

The PowerPC is a bi-endian processor that supports both big-endian and little-endian modes. The bi-endian architecture enables software developers to choose either mode when migrating operating systems and applications from other machines. The operating system establishes the endian mode in which processes execute. Once a mode is selected, all subsequent memory loads and stores are determined by the memory-addressing model of that mode. To support this hardware feature, 2 bits are maintained in the machine state register (MSR) maintained by the operating system as part of the process state. One bit specifies the endian mode in which the kernel runs; the other specifies the processor's current operating mode. Thus, mode can be changed on a per-process basis.

Bit Ordering

In ordering the bits within a byte, we are immediately faced with two questions:

1. Do you count the first bit as bit zero or as bit one?
2. Do you assign the lowest bit number to the byte's least significant bit (little endian) or to the bytes most significant bit (big endian)?

These questions are not answered in the same way on all machines. Indeed, on some machines, the answers are different in different circumstances. Furthermore, the choice of big- or little-endian bit ordering within a byte is not always consistent with big- or little-endian ordering of bytes within a multibyte scalar. The programmer needs to be concerned with these issues when manipulating individual bits.

Another area of concern is when data are transmitted over a bit-serial line. When an individual byte is transmitted, does the system transmit the most significant bit first or the least significant bit first? The designer must make certain that incoming bits are handled properly. For a discussion of this issue, see [JAME90].

CHAPTER 13

Instruction Sets: Addressing Modes and Formats

13.1 Addressing Modes
- Immediate Addressing
- Direct Addressing
- Indirect Addressing
- Register Addressing
- Register Indirect Addressing
- Displacement Addressing
- Stack Addressing

13.2 x86 and ARM Addressing Modes
- x86 Addressing Modes
- ARM Addressing Modes

13.3 Instruction Formats
- Instruction Length
- Allocation of Bits
- Variable-Length Instructions

13.4 x86 and ARM Instruction Formats
- x86 Instruction Formats
- ARM Instruction Formats

13.5 Assembly Language

13.6 Key Terms, Review Questions, and Problems

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Describe the various types of addressing modes common in instruction sets.
- Present an overview of x86 and ARM addressing modes.
- Summarize the issues and trade-offs involved in designing an **instruction format**.
- Present an overview of x86 and ARM instruction formats.
- Understand the distinction between machine language and assembly language.

In Chapter 12, we focused on *what* an instruction set does. Specifically, we examined the types of operands and operations that may be specified by machine instructions. This chapter turns to the question of *how* to specify the operands and operations of instructions. Two issues arise. First, how is the address of an operand specified, and second, how are the bits of an instruction organized to define the operand addresses and operation of that instruction?

13.1 ADDRESSING MODES

The address field or fields in a typical instruction format are relatively small. We would like to be able to reference a large range of locations in main memory or, for some systems, virtual memory. To achieve this objective, a variety of addressing techniques has been employed. They all involve some trade-off between address range and/or addressing flexibility, on the one hand, and the number of memory references in the instruction and/or the complexity of address calculation, on the other. In this section, we examine the most common addressing techniques, or modes:

- Immediate
- Direct
- Indirect
- Register
- Register indirect
- Displacement
- Stack

These modes are illustrated in Figure 13.1. In this section, we use the following notation:

A = contents of an address field in the instruction
R = contents of an address field in the instruction that refers to a register
EA = actual (effective) address of the location containing the referenced operand
(X) = contents of memory location X or register X

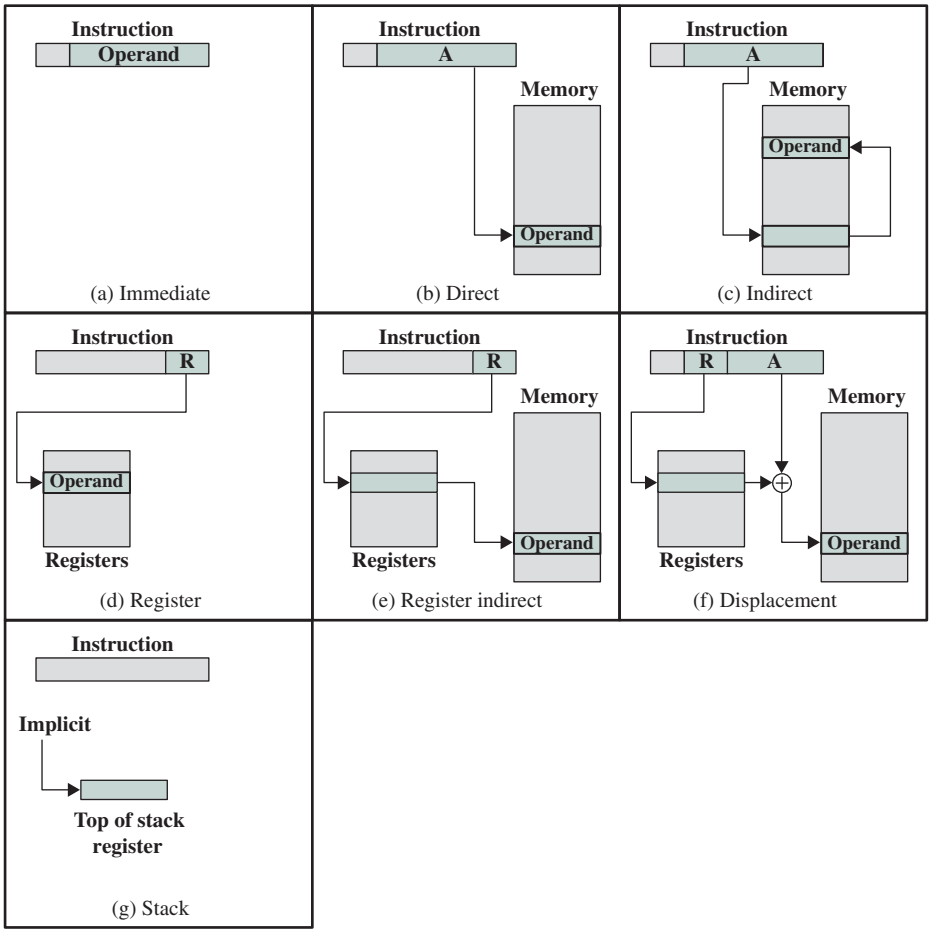


Figure 13.1 Addressing Modes

Table 13.1 indicates the address calculation performed for each addressing mode.

Before beginning this discussion, two comments need to be made. First, virtually all computer architectures provide more than one of these addressing modes. The question arises as to how the processor can determine which address mode is being used in a particular instruction. Several approaches are taken. Often, different opcodes will use different addressing modes. Also, one or more bits in the instruction format can be used as a *mode field.* The value of the mode field determines which addressing mode is to be used.

The second comment concerns the interpretation of the effective address (EA). In a system without virtual memory, the **effective address** will be either a main memory address or a register. In a virtual memory system, the effective address is a virtual address or a register. The actual mapping to a physical address is a function of the memory management unit (MMU) and is invisible to the programmer.

Table 13.1 Basic Addressing Modes

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

Immediate Addressing

The simplest form of addressing is **immediate addressing**, in which the operand value is present in the instruction

$$\text{Operand} = \text{A}$$

This mode can be used to define and use constants or set initial values of variables. Typically, the number will be stored in twos complement form; the leftmost bit of the operand field is used as a sign bit. When the operand is loaded into a data register, the sign bit is extended to the left to the full data **word** size. In some cases, the immediate binary value is interpreted as an unsigned nonnegative integer.

The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand, thus saving one memory or cache cycle in the instruction cycle. The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the word length.

Direct Addressing

A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

$$\text{EA} = \text{A}$$

The technique was common in earlier generations of computers but is not common on contemporary architectures. It requires only one memory reference and no special calculation. The obvious limitation is that it provides only a limited address space.

Indirect Addressing

With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is known as **indirect addressing**:

$$\text{EA} = (\text{A})$$

As defined earlier, the parentheses are to be interpreted as meaning *contents of*. The obvious advantage of this approach is that for a word length of N, an address space of 2^N is now available. The disadvantage is that instruction execution requires two memory references to fetch the operand: one to get its address and a second to get its value.

Although the number of words that can be addressed is now equal to 2^N, the number of different effective addresses that may be referenced at any one time is limited to 2^K, where K is the length of the address field. Typically, this is not a burdensome restriction, and it can be an asset. In a virtual memory environment, all the effective address locations can be confined to page 0 of any process. Because the address field of an instruction is small, it will naturally produce low-numbered direct addresses, which would appear in page 0. (The only restriction is that the page size must be greater than or equal to 2^K.) When a process is active, there will be repeated references to page 0, causing it to remain in real memory. Thus, an indirect memory reference will involve, at most, one page fault rather than two.

A rarely used variant of indirect addressing is multilevel or cascaded indirect addressing:

$$\text{EA} = (\ldots (\text{A}) \ldots)$$

In this case, one bit of a full-word address is an indirect flag (I). If the I bit is 0, then the word contains the EA. If the I bit is 1, then another level of indirection is invoked. There does not appear to be any particular advantage to this approach, and its disadvantage is that three or more memory references could be required to fetch an operand.

Register Addressing

Register addressing is similar to direct addressing. The only difference is that the address field refers to a register rather than a main memory address:

$$\text{EA} = \text{R}$$

To clarify, if the contents of a register address field in an instruction is 5, then register R5 is the intended address, and the operand value is contained in R5. Typically, an address field that references registers will have from 3 to 5 bits, so that a total of from 8 to 32 general-purpose registers can be referenced.

The advantages of register addressing are that (1) only a small address field is needed in the instruction, and (2) no time-consuming memory references are required. As was discussed in Chapter 4, the memory access time for a register internal to the processor is much less than that for a main memory address. The disadvantage of register addressing is that the address space is very limited.

If register addressing is heavily used in an instruction set, this implies that the processor registers will be heavily used. Because of the severely limited number of registers (compared with main memory locations), their use in this fashion makes sense only if they are employed efficiently. If every operand is brought into a register from main memory, operated on once, and then returned to main memory, then a wasteful intermediate step has been added. If, instead, the operand in a register remains in use for multiple operations, then a real savings is achieved. An example is the intermediate result in a calculation. In particular, suppose that the algorithm

for twos complement multiplication were to be implemented in software. The location labeled A in the flowchart (Figure 10.12) is referenced many times and should be implemented in a register rather than a main memory location.

It is up to the programmer or compiler to decide which values should remain in registers and which should be stored in main memory. Most modern processors employ multiple general-purpose registers, placing a burden for efficient execution on the assembly-language programmer (e.g., compiler writer).

Register Indirect Addressing

Just as register addressing is analogous to direct addressing, **register indirect addressing** is analogous to indirect addressing. In both cases, the only difference is whether the address field refers to a memory location or a register. Thus, for register indirect address,

$$EA = (R)$$

The advantages and limitations of register indirect addressing are basically the same as for indirect addressing. In both cases, the address space limitation (limited range of addresses) of the address field is overcome by having that field refer to a word-length location containing an address. In addition, register indirect addressing uses one less memory reference than indirect addressing.

Displacement Addressing

A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing. It is known by a variety of names depending on the context of its use, but the basic mechanism is the same. We will refer to this as **displacement addressing**:

$$EA = A + (R)$$

Displacement addressing requires that the instruction have two address fields, at least one of which is explicit. The value contained in one address field (value = A) is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address.

We will describe three of the most common uses of displacement addressing:

- Relative addressing
- Base-register addressing
- Indexing

RELATIVE ADDRESSING For relative addressing, also called PC-relative addressing, the implicitly referenced register is the program counter (PC). That is, the next instruction address is added to the address field to produce the EA. Typically, the address field is treated as a twos complement number for this operation. Thus, the effective address is a displacement relative to the address of the instruction.

Relative addressing exploits the concept of locality that was discussed in Chapters 4 and 8. If most memory references are relatively near to the instruction being executed, then the use of relative addressing saves address bits in the instruction.

BASE–REGISTER ADDRESSING For **base-register addressing**, the interpretation is the following: The referenced register contains a main memory address, and the address field contains a displacement (usually an unsigned integer representation) from that address. The register reference may be explicit or implicit.

Base-register addressing also exploits the locality of memory references. It is a convenient means of implementing segmentation, which was discussed in Chapter 8. In some implementations, a single segment-base register is employed and is used implicitly. In others, the programmer may choose a register to hold the base address of a segment, and the instruction must reference it explicitly. In this latter case, if the length of the address field is K and the number of possible registers is N, then one instruction can reference any one of N areas of 2^K words.

INDEXING For indexing, the interpretation is typically the following: The address field references a main memory address, and the referenced register contains a positive displacement from that address. Note that this usage is just the opposite of the interpretation for base-register addressing. Of course, it is more than just a matter of user interpretation. Because the address field is considered to be a memory address in indexing, it generally contains more bits than an address field in a comparable base-register instruction. Also, we will see that there are some refinements to indexing that would not be as useful in the base-register context. Nevertheless, the method of calculating the EA is the same for both base-register addressing and indexing, and in both cases the register reference is sometimes explicit and sometimes implicit (for different processor types).

An important use of indexing is to provide an efficient mechanism for performing iterative operations. Consider, for example, a list of numbers stored starting at location A. Suppose that we would like to add 1 to each element on the list. We need to fetch each value, add 1 to it, and store it back. The sequence of effective addresses that we need is A, A + 1, A + 2, . . . , up to the last location on the list. With indexing, this is easily done. The value A is stored in the instruction's address field, and the chosen register, called an *index register*, is initialized to 0. After each operation, the index register is incremented by 1.

Because index registers are commonly used for such iterative tasks, it is typical that there is a need to increment or decrement the index register after each reference to it. Because this is such a common operation, some systems will automatically do this as part of the same instruction cycle. This is known as **autoindexing**. If certain registers are devoted exclusively to indexing, then autoindexing can be invoked implicitly and automatically. If general-purpose registers are used, the autoindex operation may need to be signaled by a bit in the instruction. Autoindexing using increment can be depicted as follows.

$$\text{EA} = \text{A} + (\text{R})$$
$$(\text{R}) \leftarrow (\text{R}) + 1$$

In some machines, both indirect addressing and indexing are provided, and it is possible to employ both in the same instruction. There are two possibilities: the indexing is performed either before or after the indirection.

If indexing is performed after the indirection, it is termed **postindexing**:

$$\text{EA} = (\text{A}) + (\text{R})$$

First, the contents of the address field are used to access a memory location containing a direct address. This address is then indexed by the register value. This technique is useful for accessing one of a number of blocks of data of a fixed format. For example, it was described in Chapter 8 that the operating system needs to employ a process control block for each process. The operations performed are the same regardless of which block is being manipulated. Thus, the addresses in the instructions that reference the block could point to a location (value = A) containing a variable pointer to the start of a process control block. The index register contains the displacement within the block.

With **preindexing**, the indexing is performed before the indirection:

$$EA = (A + (R))$$

An address is calculated as with simple indexing. In this case, however, the calculated address contains not the operand, but the address of the operand. An example of the use of this technique is to construct a multiway branch table. At a particular point in a program, there may be a branch to one of a number of locations depending on conditions. A table of addresses can be set up starting at location A. By indexing into this table, the required location can be found.

Typically, an instruction set will not include both preindexing and postindexing.

Stack Addressing

The final addressing mode that we consider is stack addressing. As defined in Appendix I, a stack is a linear array of locations. It is sometimes referred to as a *pushdown list* or *last-in-first-out queue*. The stack is a reserved block of locations. Items are appended to the top of the stack so that, at any given time, the block is partially filled. Associated with the stack is a pointer whose value is the address of the top of the stack. Alternatively, the top two elements of the stack may be in processor registers, in which case the stack pointer references the third element of the stack. The stack pointer is maintained in a register. Thus, references to stack locations in memory are in fact register indirect addresses.

The stack mode of addressing is a form of implied addressing. The machine instructions need not include a memory reference but implicitly operate on the top of the stack.

13.2 x86 AND ARM ADDRESSING MODES

x86 Addressing Modes

Recall from Figure 8.21 that the x86 address translation mechanism produces an address, called a virtual or effective address, that is an offset into a segment. The sum of the starting address of the segment and the effective address produces a linear address. If paging is being used, this linear address must pass through a page-translation mechanism to produce a physical address. In what follows, we ignore this last step because it is transparent to the instruction set and to the programmer.

The x86 is equipped with a variety of addressing modes intended to allow the efficient execution of high-level languages. Figure 13.2 indicates the logic

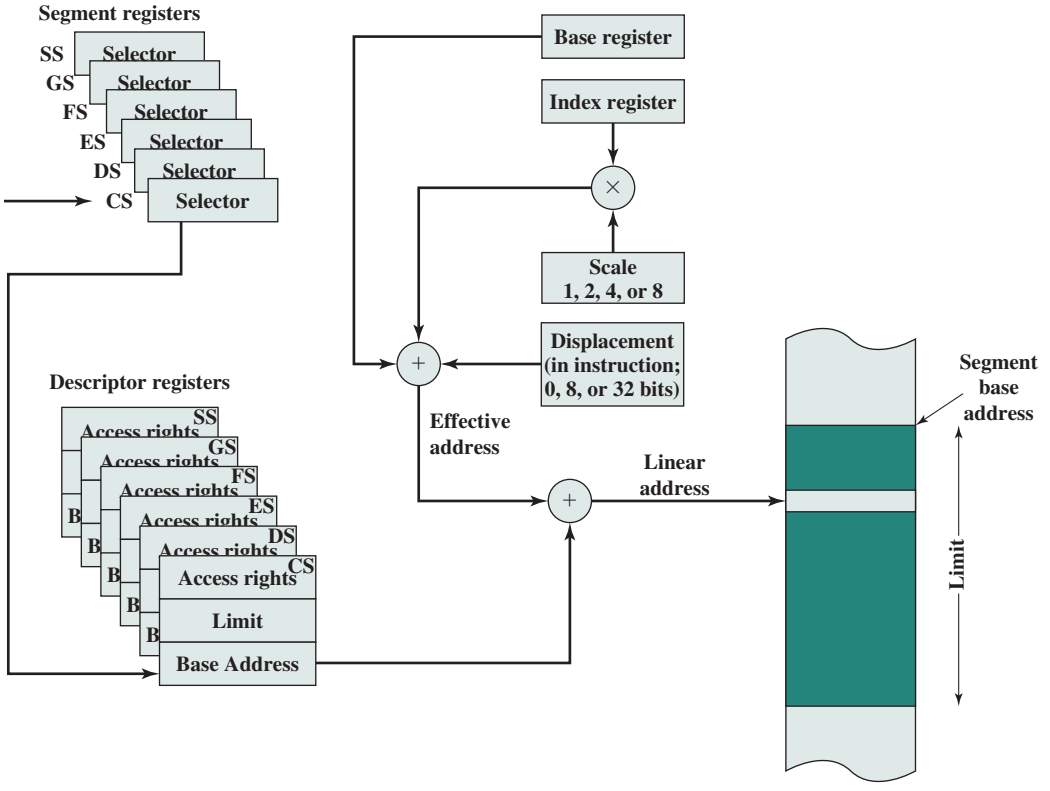


Figure 13.2 x86 Addressing Mode Calculation

involved. The segment register determines the segment that is the subject of the reference. There are six segment registers; the one being used for a particular reference depends on the context of execution and the instruction. Each segment register holds an index into the segment descriptor table (Figure 8.20), which holds the starting address of the corresponding segments. Associated with each user-visible segment register is a segment descriptor register (not programmer visible), which records the access rights for the segment as well as the starting address and limit (length) of the segment. In addition, there are two registers that may be used in constructing an address: the base register and the index register.

Table 13.2 lists the x86 addressing modes. Let us consider each of these in turn.

For the **immediate mode**, the operand is included in the instruction. The operand can be a byte, word, or doubleword of data.

For **register operand mode**, the operand is located in a register. For general instructions, such as data transfer, arithmetic, and logical instructions, the operand can be one of the 32-bit general registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP), one of the 16-bit general registers (AX, BX, CX, DX, SI, DI, SP, BP), or one of the 8-bit general registers (AH, BH, CH, DH, AL, BL, CL, DL). There are also some instructions that reference the segment selector registers (CS, DS, ES, SS, FS, GS).

Table 13.2 x86 Addressing Modes

Mode	Algorithm
Immediate	Operand = A
Register Operand	LA = R
Displacement	LA = (SR) + A
Base	LA = (SR) + (B)
Base with Displacement	LA = (SR) + (B) + A
Scaled Index with Displacement	LA = (SR) + (I) × S + A
Base with Index and Displacement	LA = (SR) + (B) + (I) + A
Base with Scaled Index and Displacement	LA = (SR) + (I) × S + (B) + A
Relative	LA = (PC) + A

LA = linear address
(X) = contents of X
SR = segment register
PC = program counter
A = contents of an address field in the instruction
R = register
B = base register
I = index register
S = scaling factor

The remaining addressing modes reference locations in memory. The memory location must be specified in terms of the segment containing the location and the offset from the beginning of the segment. In some cases, a segment is specified explicitly; in others, the segment is specified by simple rules that assign a segment by default.

In the **displacement mode**, the operand's offset (the effective address of Figure 13.2) is contained as part of the instruction as an 8-, 16-, or 32-bit displacement. With segmentation, all addresses in instructions refer merely to an offset in a segment. The displacement addressing mode is found on few machines because, as mentioned earlier, it leads to long instructions. In the case of the x86, the displacement value can be as long as 32 bits, making for a 6-byte instruction. Displacement addressing can be useful for referencing global variables.

The remaining addressing modes are indirect, in the sense that the address portion of the instruction tells the processor where to look to find the address. The **base mode** specifies that one of the 8-, 16-, or 32-bit registers contains the effective address. This is equivalent to what we have referred to as register indirect addressing.

In the **base with displacement mode**, the instruction includes a displacement to be added to a base register, which may be any of the general-purpose registers. Examples of uses of this mode are as follows:

- Used by a compiler to point to the start of a local variable area. For example, the base register could point to the beginning of a stack frame, which contains the local variables for the corresponding procedure.
- Used to index into an array when the element size is not 1, 2, 4, or 8 bytes and which therefore cannot be indexed using an index register. In this case, the displacement points to the beginning of the array, and the base register holds the results of a calculation to determine the offset to a specific element within the array.

- Used to access a field of a record. The base register points to the beginning of the record, while the displacement is an offset to the field.

In the **scaled index with displacement mode**, the instruction includes a displacement to be added to a register, in this case called an index register. The index register may be any of the general-purpose registers except the one called ESP, which is generally used for stack processing. In calculating the effective address, the contents of the index register are multiplied by a scaling factor of 1, 2, 4, or 8, and then added to a displacement. This mode is very convenient for indexing arrays. A scaling factor of 2 can be used for an array of 16-bit integers. A scaling factor of 4 can be used for 32-bit integers or floating-point numbers. Finally, a scaling factor of 8 can be used for an array of double-precision floating-point numbers.

The **base with index and displacement mode** sums the contents of the base register, the index register, and a displacement to form the effective address. Again, the base register can be any general-purpose register and the index register can be any general-purpose register except ESP. As an example, this addressing mode could be used for accessing a local array on a stack frame. This mode can also be used to support a two-dimensional array; in this case, the displacement points to the beginning of the array, and each register handles one dimension of the array.

The **based scaled index with displacement mode** sums the contents of the index register multiplied by a scaling factor, the contents of the base register, and the displacement. This is useful if an array is stored in a stack frame; in this case, the array elements would be 2, 4, or 8 bytes each in length. This mode also provides efficient indexing of a two-dimensional array when the array elements are 2, 4, or 8 bytes in length.

Finally, **relative addressing** can be used in transfer-of-control instructions. A displacement is added to the value of the program counter, which points to the next instruction. In this case, the displacement is treated as a signed byte, word, or doubleword value, and that value either increases or decreases the address in the program counter.

ARM Addressing Modes

Typically, a RISC machine, unlike a CISC machine, uses a simple and relatively straightforward set of addressing modes. The ARM architecture departs somewhat from this tradition by providing a relatively rich set of addressing modes. These modes are most conveniently classified with respect to the type of instruction.[1]

LOAD/STORE ADDRESSING Load and store instructions are the only instructions that reference memory. This is always done indirectly through a base register plus offset. There are three alternatives with respect to indexing (Figure 13.3):

- **Offset:** For this addressing method, **indexing** is not used. An offset value is added to or subtracted from the value in the base register to form the memory address. As an example Figure 13.3a illustrates this method with the assembly language instruction `STRB r0, [r1, #12]`. This is the store byte instruction.

[1]As with our discussion of x86 addressing, we ignore the translation from virtual to physical address in the following discussion.

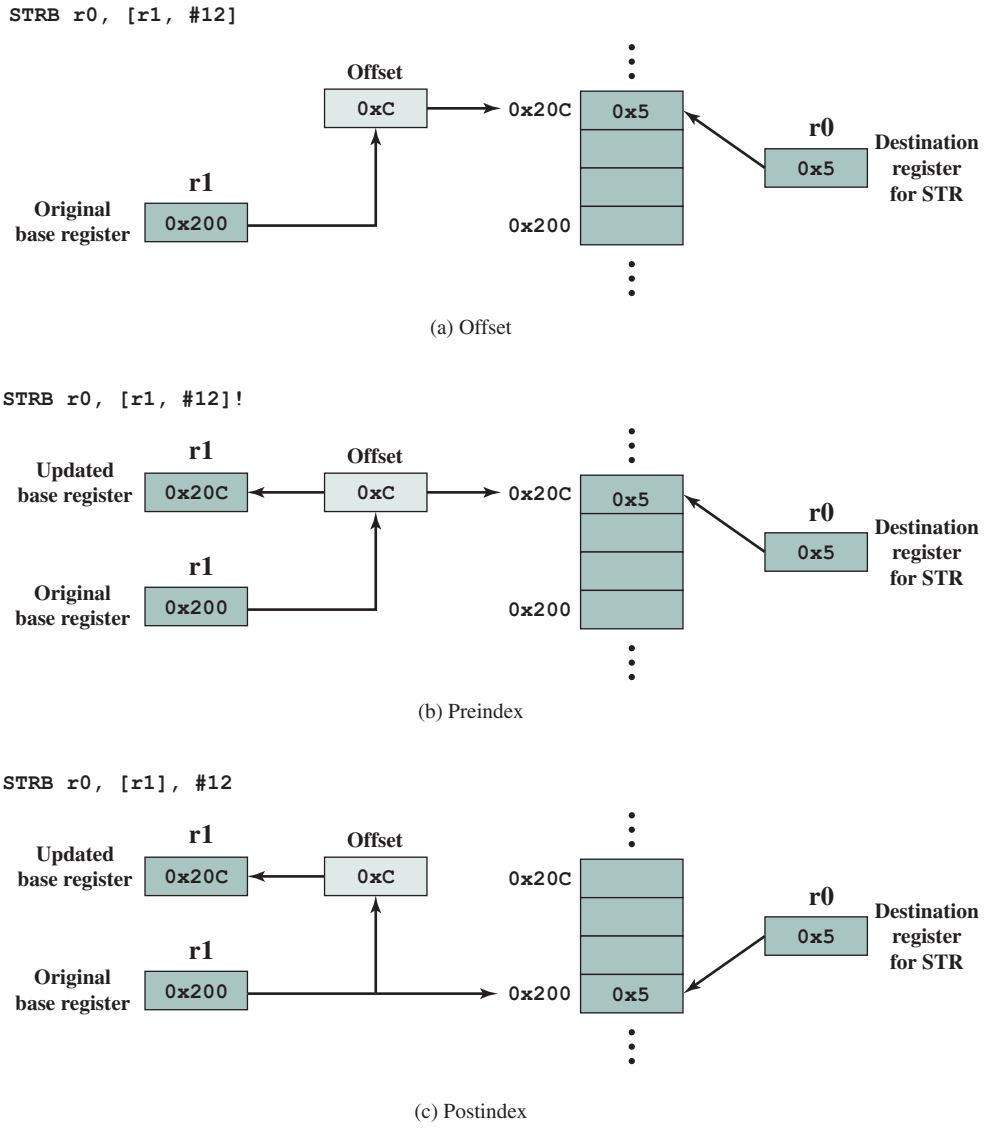


Figure 13.3 ARM Indexing Methods

In this case the base address is in register r1 and the displacement is an immediate value of decimal 12. The resulting address (base plus offset) is the location where the least significant byte from r0 is to be stored.

- **Preindex:** The memory address is formed in the same way as for offset addressing. The memory address is also written back to the base register. In other words, the base register value is incremented or decremented by the offset value. Figure 13.3b illustrates this method with the assembly language instruction `STRB r0, [r1, #12]!`. The exclamation point signifies preindexing.

- **Postindex:** The memory address is the base register value. An offset is added to or subtracted from the base register value and the result is written back to the base register. Figure 13.3c illustrates this method with the assembly language instruction `STRB r0, [r1], #12`.

Note that what ARM refers to as a base register acts as an index register for preindex and postindex addressing. The offset value can either be an immediate value stored in the instruction or it can be in another register. If the offset value is in a register, another useful feature is available: scaled register addressing. The value in the offset register is scaled by one of the shift operators: Logical Shift Left, Logical Shift Right, Arithmetic Shift Right, Rotate Right, or Rotate Right Extended (which includes the carry bit in the rotation). The amount of the shift is specified as an immediate value in the instruction.

DATA PROCESSING INSTRUCTION ADDRESSING Data processing instructions use either register addressing or a mixture of register and immediate addressing. For register addressing, the value in one of the register operands may be scaled using one of the five shift operators defined in the preceding paragraph.

BRANCH INSTRUCTIONS The only form of addressing for branch instructions is immediate addressing. The branch instruction contains a 24-bit value. For address calculation, this value is shifted left 2 bits, so that the address is on a word boundary. Thus the effective address range is ± 32 MB from the program counter.

LOAD/STORE MULTIPLE ADDRESSING Load Multiple instructions load a subset (possibly all) of the general-purpose registers from memory. Store Multiple instructions store a subset (possibly all) of the general-purpose registers to memory. The list of registers for the load or store is specified in a 16-bit field in the instruction with each bit corresponding to one of the 16 registers. Load and Store Multiple addressing modes produce a sequential range of memory addresses. The lowest-numbered register is stored at the lowest memory address and the highest-numbered register at the highest memory address. Four addressing modes are used (Figure 13.4): increment after, increment before, decrement after, and decrement before. A base

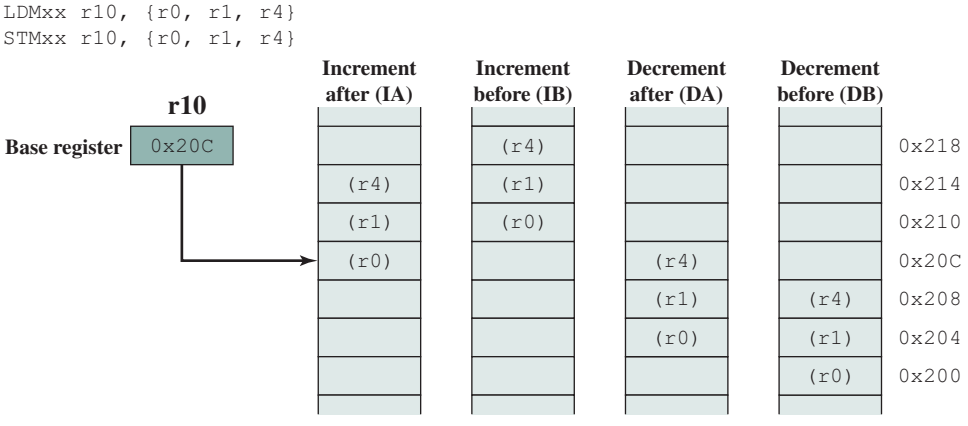


Figure 13.4 ARM Load/Store Multiple Addressing

register specifies a main memory address where register values are stored in or loaded from in ascending (increment) or descending (decrement) word locations. Incrementing or decrementing starts either before or after the first memory access.

These instructions are useful for block loads or stores, stack operations, and procedure exit sequences.

13.3 INSTRUCTION FORMATS

An instruction format defines the layout of the bits of an instruction, in terms of its constituent fields. An instruction format must include an opcode and, implicitly or explicitly, zero or more operands. Each explicit operand is referenced using one of the addressing modes described in Section 13.1. The format must, implicitly or explicitly, indicate the addressing mode for each operand. For most instruction sets, more than one instruction format is used.

The design of an instruction format is a complex art, and an amazing variety of designs have been implemented. We examine the key design issues, looking briefly at some designs to illustrate points, and then we examine the x86 and ARM solutions in detail.

Instruction Length

The most basic design issue to be faced is the instruction format length. This decision affects, and is affected by, memory size, memory organization, bus structure, processor complexity, and processor speed. This decision determines the richness and flexibility of the machine as seen by the assembly-language programmer.

The most obvious trade-off here is between the desire for a powerful instruction repertoire and a need to save space. Programmers want more opcodes, more operands, more addressing modes, and greater address range. More opcodes and more operands make life easier for the programmer, because shorter programs can be written to accomplish given tasks. Similarly, more addressing modes give the programmer greater flexibility in implementing certain functions, such as table manipulations and multiple-way branching. And, of course, with the increase in main memory size and the increasing use of virtual memory, programmers want to be able to address larger memory ranges. All of these things (opcodes, operands, addressing modes, address range) require bits and push in the direction of longer instruction lengths. But longer instruction length may be wasteful. A 64-bit instruction occupies twice the space of a 32-bit instruction but is probably less than twice as useful.

Beyond this basic trade-off, there are other considerations. Either the instruction length should be equal to the memory-transfer length (in a bus system, data-bus length) or one should be a multiple of the other. Otherwise, we will not get an integral number of instructions during a fetch cycle. A related consideration is the memory transfer rate. This rate has not kept up with increases in processor speed. Accordingly, memory can become a bottleneck if the processor can execute instructions faster than it can fetch them. One solution to this problem is to use cache memory (see Section 4.3); another is to use shorter instructions. Thus, 16-bit instructions can be fetched at twice the rate of 32-bit instructions but probably can be executed less than twice as rapidly.

A seemingly mundane but nevertheless important feature is that the instruction length should be a multiple of the character length, which is usually 8 bits, and of the length of fixed-point numbers. To see this, we need to make use of that unfortunately ill-defined word, *word* [FRAI83]. The word length of memory is, in some sense, the "natural" unit of organization. The size of a word usually determines the size of fixed-point numbers (usually the two are equal). Word size is also typically equal to, or at least integrally related to, the memory transfer size. Because a common form of data is character data, we would like a word to store an integral number of characters. Otherwise, there are wasted bits in each word when storing multiple characters, or a character will have to straddle a word boundary. The importance of this point is such that IBM, when it introduced the System/360 and wanted to employ 8-bit characters, made the wrenching decision to move from the 36-bit architecture of the scientific members of the 700/7000 series to a 32-bit architecture.

Allocation of Bits

We've looked at some of the factors that go into deciding the length of the instruction format. An equally difficult issue is how to allocate the bits in that format. The trade-offs here are complex.

For a given instruction length, there is clearly a trade-off between the number of opcodes and the power of the addressing capability. More opcodes obviously mean more bits in the opcode field. For an instruction format of a given length, this reduces the number of bits available for addressing. There is one interesting refinement to this trade-off, and that is the use of variable-length opcodes. In this approach, there is a minimum opcode length but, for some opcodes, additional operations may be specified by using additional bits in the instruction. For a fixed-length instruction, this leaves fewer bits for addressing. Thus, this feature is used for those instructions that require fewer operands and/or less powerful addressing.

The following interrelated factors go into determining the use of the addressing bits.

- **Number of addressing modes:** Sometimes an addressing mode can be indicated implicitly. For example, certain opcodes might always call for indexing. In other cases, the addressing modes must be explicit, and one or more mode bits will be needed.
- **Number of operands:** We have seen that fewer addresses can make for longer, more awkward programs (e.g., Figure 12.3). Typical instruction formats on today's machines include two operands. Each operand address in the instruction might require its own mode indicator, or the use of a mode indicator could be limited to just one of the address fields.
- **Register versus memory:** A machine must have registers so that data can be brought into the processor for processing. With a single user-visible register (usually called the accumulator), one operand address is implicit and consumes no instruction bits. However, single-register programming is awkward and requires many instructions. Even with multiple registers, only a few bits are needed to specify the register. The more that registers can be used for

operand references, the fewer bits are needed. A number of studies indicate that a total of 8 to 32 user-visible registers is desirable [LUND77, HUCK83]. Most contemporary architectures have at least 32 registers.

- **Number of register sets:** Most contemporary machines have one set of general-purpose registers, with typically 32 or more registers in the set. These registers can be used to store data and can be used to store addresses for displacement addressing. Some architectures, including that of the x86, have a collection of two or more specialized sets (such as data and displacement). One advantage of this latter approach is that, for a fixed number of registers, a functional split requires fewer bits to be used in the instruction. For example, with two sets of eight registers, only 3 bits are required to identify a register; the opcode or mode register will determine which set of registers is being referenced.
- **Address range:** For addresses that reference memory, the range of addresses that can be referenced is related to the number of address bits. Because this imposes a severe limitation, direct addressing is rarely used. With displacement addressing, the range is opened up to the length of the address register. Even so, it is still convenient to allow rather large displacements from the register address, which requires a relatively large number of address bits in the instruction.
- **Address granularity:** For addresses that reference memory rather than registers, another factor is the granularity of addressing. In a system with 16- or 32-bit words, an address can reference a word or a byte at the designer's choice. Byte addressing is convenient for character manipulation but requires, for a fixed-size memory, more address bits.

Thus, the designer is faced with a host of factors to consider and balance. How critical the various choices are is not clear. As an example, we cite one study [CRAG79] that compared various instruction format approaches, including the use of a stack, general-purpose registers, an accumulator, and only memory-to-register approaches. Using a consistent set of assumptions, no significant difference in code space or execution time was observed.

Let us briefly look at how two historical machine designs balance these various factors.

PDP-8 One of the simplest instruction designs for a general-purpose computer was for the PDP-8 [BELL78b]. The PDP-8 uses 12-bit instructions and operates on 12-bit words. There is a single general-purpose register, the accumulator.

Despite the limitations of this design, the addressing is quite flexible. Each memory reference consists of 7 bits plus two 1-bit modifiers. The memory is divided into fixed-length pages of $2^7 = 128$ words each. Address calculation is based on references to page 0 or the current page (page containing this instruction) as determined by the page bit. The second modifier bit indicates whether direct or indirect addressing is to be used. These two modes can be used in combination, so that an indirect address is a 12-bit address contained in a word of page 0 or the current page. In addition, 8 dedicated words on page 0 are autoindex "registers." When an indirect reference is made to one of these locations, preindexing occurs.

Figure 13.5 shows the PDP-8 instruction format. There are a 3-bit opcode and three types of instructions. For opcodes 0 through 5, the format is a single-address

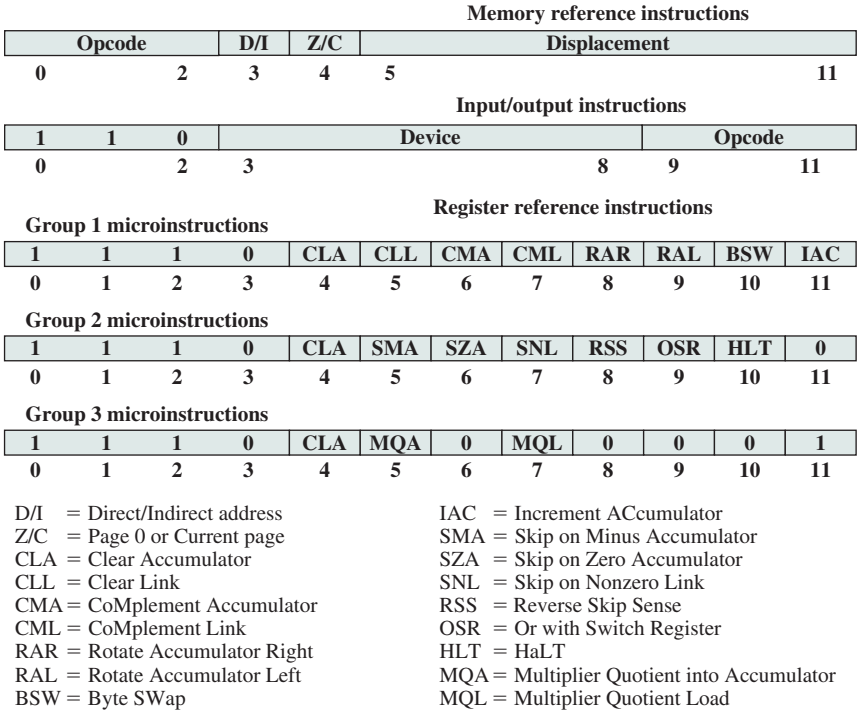


Figure 13.5 PDP-8 Instruction Formats

memory reference instruction including a page bit and an indirect bit. Thus, there are only six basic operations. To enlarge the group of operations, opcode 7 defines a register reference or *microinstruction.* In this format, the remaining bits are used to encode additional operations. In general, each bit defines a specific operation (e.g., clear accumulator), and these bits can be combined in a single instruction. The microinstruction strategy was used as far back as the PDP-1 by DEC and is, in a sense, a forerunner of today's microprogrammed machines, to be discussed in Part Four. Opcode 6 is the I/O operation; 6 bits are used to select one of 64 devices, and 3 bits specify a particular I/O command.

The PDP-8 instruction format is remarkably efficient. It supports indirect addressing, displacement addressing, and indexing. With the use of the opcode extension, it supports a total of approximately 35 instructions. Given the constraints of a 12-bit instruction length, the designers could hardly have done better.

PDP-10 A sharp contrast to the instruction set of the PDP-8 is that of the PDP-10. The PDP-10 was designed to be a large-scale time-shared system, with an emphasis on making the system easy to program, even if additional hardware expense was involved.

Among the design principles employed in designing the instruction set were the following [BELL78c]:

- **Orthogonality:** Orthogonality is a principle by which two variables are independent of each other. In the context of an instruction set, the term indicates

that other elements of an instruction are independent of (not determined by) the opcode. The PDP-10 designers use the term to describe the fact that an address is always computed in the same way, independent of the opcode. This is in contrast to many machines, where the address mode sometimes depends implicitly on the operator being used.

- **Completeness:** Each arithmetic data type (integer, fixed-point, floating-point) should have a complete and identical set of operations.
- **Direct addressing:** Base plus displacement addressing, which places a memory organization burden on the programmer, was avoided in favor of direct addressing.

Each of these principles advances the main goal of ease of programming.

The PDP-10 has a 36-bit word length and a 36-bit instruction length. The fixed instruction format is shown in Figure 13.6. The opcode occupies 9 bits, allowing up to 512 operations. In fact, a total of 365 different instructions are defined. Most instructions have two addresses, one of which is one of 16 general-purpose registers. Thus, this operand reference occupies 4 bits. The other operand reference starts with an 18-bit memory address field. This can be used as an immediate operand or a memory address. In the latter usage, both indexing and indirect addressing are allowed. The same general-purpose registers are also used as index registers.

A 36-bit instruction length is true luxury. There is no need to do clever things to get more opcodes; a 9-bit opcode field is more than adequate. Addressing is also straightforward. An 18-bit address field makes direct addressing desirable. For memory sizes greater than 2^{18}, indirection is provided. For the ease of the programmer, indexing is provided for table manipulation and iterative programs. Also, with an 18-bit operand field, immediate addressing becomes attractive.

The PDP-10 instruction set design does accomplish the objectives listed earlier [LUND77]. It eases the task of the programmer or compiler at the expense of an inefficient utilization of space. This was a conscious choice made by the designers and therefore cannot be faulted as poor design.

Variable-Length Instructions

The examples we have looked at so far have used a single fixed instruction length, and we have implicitly discussed trade-offs in that context. But the designer may choose instead to provide a variety of instruction formats of different lengths. This tactic makes it easy to provide a large repertoire of opcodes, with different opcode lengths. Addressing can be more flexible, with various combinations of register and memory references plus addressing modes. With variable-length instructions, these many variations can be provided efficiently and compactly.

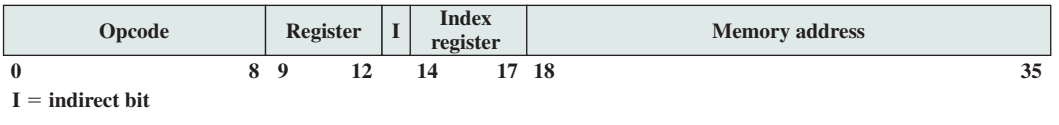


Figure 13.6 PDP-10 Instruction Format

The principal price to pay for variable-length instructions is an increase in the complexity of the processor. Falling hardware prices, the use of microprogramming (discussed in Part Four), and a general increase in understanding the principles of processor design have all contributed to making this a small price to pay. However, we will see that RISC and superscalar machines can exploit the use of fixed-length instructions to provide improved performance.

The use of variable-length instructions does not remove the desirability of making all of the instruction lengths integrally related to the word length. Because the processor does not know the length of the next instruction to be fetched, a typical strategy is to fetch a number of bytes or words equal to at least the longest possible instruction. This means that sometimes multiple instructions are fetched. However, as we shall see in Chapter 14, this is a good strategy to follow in any case.

PDP-11 The PDP-11 was designed to provide a powerful and flexible instruction set within the constraints of a 16-bit minicomputer [BELL70].

The PDP-11 employs a set of eight 16-bit general-purpose registers. Two of these registers have additional significance: one is used as a stack pointer for special-purpose stack operations, and one is used as the program counter, which contains the address of the next instruction.

Figure 13.7 shows the PDP-11 instruction formats. Thirteen different formats are used, encompassing zero-, one-, and two-address instruction types. The opcode can vary from 4 to 16 bits in length. Register references are 6 bits in length. Three bits identify the register, and the remaining 3 bits identify the addressing mode. The PDP-11 is endowed with a rich set of addressing modes. One advantage of linking the addressing mode to the operand rather than the opcode, as is sometimes done, is that any addressing mode can be used with any opcode. As was mentioned, this independence is referred to as *orthogonality*.

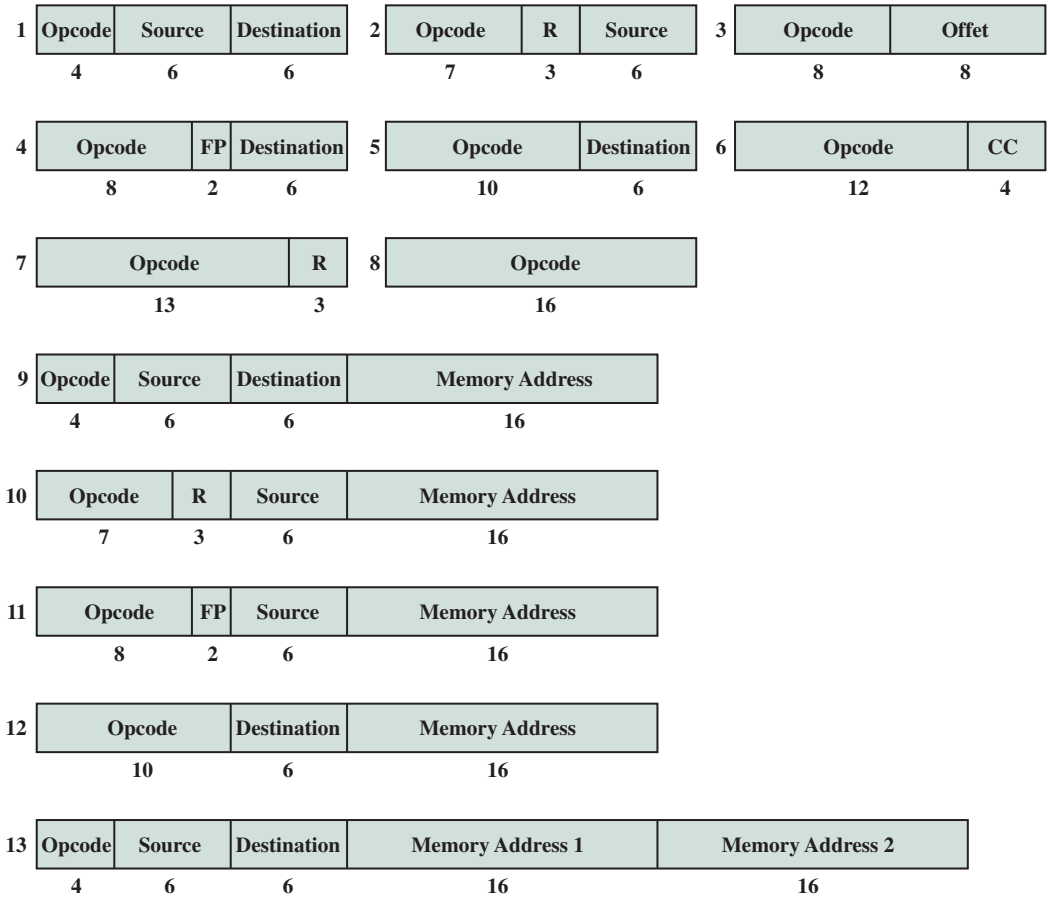
PDP-11 instructions are usually one word (16 bits) long. For some instructions, one or two memory addresses are appended, so that 32-bit and 48-bit instructions are part of the repertoire. This provides for further flexibility in addressing.

The PDP-11 instruction set and addressing capability are complex. This increases both hardware cost and programming complexity. The advantage is that more efficient or compact programs can be developed.

VAX Most architectures provide a relatively small number of fixed instruction formats. This can cause two problems for the programmer. First, addressing mode and opcode are not orthogonal. For example, for a given operation, one operand must come from a register and another from memory, or both from registers, and so on. Second, only a limited number of operands can be accommodated: typically up to two or three. Because some operations inherently require more operands, various strategies must be used to achieve the desired result using two or more instructions.

To avoid these problems, two criteria were used in designing the VAX instruction format [STRE78]:

1. All instructions should have the "natural" number of operands.
2. All operands should have the same generality in specification.



Numbers below fields indicate bit length.
Source and destination each contain a 3-bit addressing mode field and a 3-bit register number.
FP indicates one of four floating-point registers.
R indicates one of the general-purpose registers.
CC is the condition code field.

Figure 13.7 Instruction Formats for the PDP-11

The result is a highly variable instruction format. An instruction consists of a 1- or 2-byte opcode followed by from zero to six operand specifiers, depending on the opcode. The minimal instruction length is 1 byte, and instructions up to 37 bytes can be constructed. Figure 13.8 gives a few examples.

The VAX instruction begins with a 1-byte opcode. This suffices to handle most VAX instructions. However, as there are over 300 different instructions, 8 bits are not enough. The hexadecimal codes FD and FF indicate an extended opcode, with the actual opcode being specified in the second byte.

The remainder of the instruction consists of up to six operand specifiers. An operand specifier is, at minimum, a 1-byte format in which the leftmost 4 bits are the address mode specifier. The only exception to this rule is the literal mode,

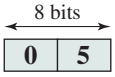
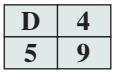
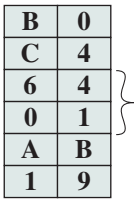
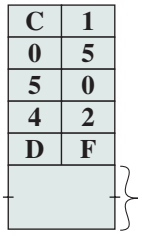
Hexadecimal Format	Explanation	Assembler Notation and Description
	Opcode for RSB	RSB Return from subroutine
	Opcode for CLRL Register R9	CLRL R9 Clear register R9
	Opcode for MOVW Word displacement mode, Register R4 356 in hexadecimal Byte displacement mode, Register R11 25 in hexadecimal	MOVW 356(R4), 25(R11) Move a word from address that is 356 plus contents of R4 to address that is 25 plus contents of R11
	Opcode for ADDL3 Short literal 5 Register mode R0 Index prefix R2 Indirect word relative (displacement from PC) Amount of displacement from PC relative to location A	ADDL3 #5, R0, @A[R2] Add 5 to a 32-bit integer in R0 and store the result in location whose address is sum of A and 4 times the contents of R2

Figure 13.8 Example of VAX Instructions

which is signaled by the pattern 00 in the leftmost 2 bits, leaving space for a 6-bit literal. Because of this exception, a total of 12 different addressing modes can be specified.

An operand specifier often consists of just one byte, with the rightmost 4 bits specifying one of 16 general-purpose registers. The length of the operand specifier can be extended in one of two ways. First, a constant value of one or more bytes may immediately follow the first byte of the operand specifier. An example of this is the displacement mode, in which an 8-, 16-, or 32-bit displacement is used. Second, an index mode of addressing may be used. In this case, the first byte of the operand specifier consists of the 4-bit addressing mode code of 0100 and a 4-bit index register identifier. The remainder of the operand specifier consists of the base address specifier, which may itself be one or more bytes in length.

The reader may be wondering, as the author did, what kind of instruction requires six operands. Surprisingly, the VAX has a number of such instructions. Consider

ADDP6 OP1, OP2, OP3, OP4, OP5, OP6

This instruction adds two packed decimal numbers. OP1 and OP2 specify the length and starting address of one decimal string; OP3 and OP4 specify a second string. These two strings are added and the result is stored in the decimal string whose length and starting location are specified by OP5 and OP6.

The VAX instruction set provides for a wide variety of operations and addressing modes. This gives a programmer, such as a compiler writer, a very powerful and flexible tool for developing programs. In theory, this should lead to efficient machine-language compilations of high-level language programs and, in general, to effective and efficient use of processor resources. The penalty to be paid for these benefits is the increased complexity of the processor compared with a processor with a simpler instruction set and format.

We return to these matters in Chapter 15, where we examine the case for very simple instruction sets.

13.4 x86 AND ARM INSTRUCTION FORMATS

x86 Instruction Formats

The x86 is equipped with a variety of instruction formats. Of the elements described in this subsection, only the opcode field is always present. Figure 13.9 illustrates the general instruction format. Instructions are made up of from zero to four optional instruction prefixes, a 1- or 2-byte opcode, an optional address specifier (which consists of the ModR/M byte and the Scale Index Base byte) an optional displacement, and an optional immediate field.

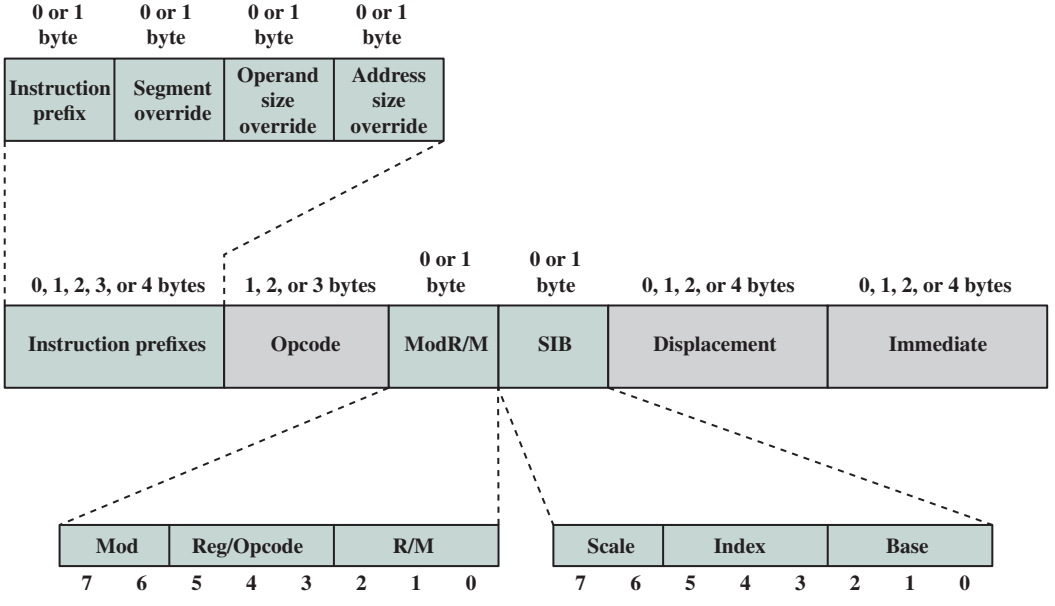


Figure 13.9 x86 Instruction Format

Let us first consider the prefix bytes:

- **Instruction prefixes:** The instruction prefix, if present, consists of the LOCK prefix or one of the repeat prefixes. The LOCK prefix is used to ensure exclusive use of shared memory in multiprocessor environments. The repeat prefixes specify repeated operation of a string, which enables the x86 to process strings much faster than with a regular software loop. There are five different repeat prefixes: REP, REPE, REPZ, REPNE, and REPNZ. When the absolute REP prefix is present, the operation specified in the instruction is executed repeatedly on successive elements of the string; the number of repetitions is specified in register CX. The conditional REP prefix causes the instruction to repeat until the count in CX goes to zero or until the condition is met.
- **Segment override:** Explicitly specifies which segment register an instruction should use, overriding the default segment-register selection generated by the x86 for that instruction.
- **Operand size:** An instruction has a default operand size of 16 or 32 bits, and the operand prefix switches between 32-bit and 16-bit operands.
- **Address size:** The processor can address memory using either 16- or 32-bit addresses. The address size determines the displacement size in instructions and the size of address offsets generated during effective address calculation. One of these sizes is designated as default, and the address size prefix switches between 32-bit and 16-bit address generation.

The instruction itself includes the following fields:

- **Opcode:** The opcode field is 1, 2, or 3 bytes in length. The opcode may also include bits that specify if data is byte- or full-size (16 or 32 bits depending on context), direction of data operation (to or from memory), and whether an immediate data field must be sign extended.
- **ModR/M:** This byte, and the next, provide addressing information. The ModR/M byte specifies whether an operand is in a register or in memory; if it is in memory, then fields within the byte specify the addressing mode to be used. The ModR/M byte consists of three fields: The Mod field (2 bits) combines with the R/M field to form 32 possible values: 8 registers and 24 indexing modes; the Reg/Opcode field (3 bits) specifies either a register number or three more bits of opcode information; the R/M field (3 bits) can specify a register as the location of an operand, or it can form part of the addressing-mode encoding in combination with the Mod field.
- **SIB:** Certain encoding of the ModR/M byte specifies the inclusion of the SIB byte to specify fully the addressing mode. The SIB byte consists of three fields: The Scale field (2 bits) specifies the scale factor for scaled indexing; the Index field (3 bits) specifies the index register; the Base field (3 bits) specifies the base register.
- **Displacement:** When the addressing-mode specifier indicates that a displacement is used, an 8-, 16-, or 32-bit signed integer displacement field is added.
- **Immediate:** Provides the value of an 8-, 16-, or 32-bit operand.

Several comparisons may be useful here. In the x86 format, the addressing mode is provided as part of the opcode sequence rather than with each operand.

Because only one operand can have address-mode information, only one memory operand can be referenced in an instruction. In contrast, the VAX carries the address-mode information with each operand, allowing memory-to-memory operations. The x86 instructions are therefore more compact. However, if a memory-to-memory operation is required, the VAX can accomplish this in a single instruction.

The x86 format allows the use of not only 1-byte, but also 2-byte and 4-byte offsets for indexing. Although the use of the larger index offsets results in longer instructions, this feature provides needed flexibility. For example, it is useful in addressing large arrays or large stack frames. In contrast, the IBM S/370 instruction format allows offsets no greater than 4 Kbytes (12 bits of offset information), and the offset must be positive. When a location is not in reach of this offset, the compiler must generate extra code to generate the needed address. This problem is especially apparent in dealing with stack frames that have local variables occupying in excess of 4 Kbytes. As [DEWA90] puts it, "generating code for the 370 is so painful as a result of that restriction that there have even been compilers for the 370 that simply chose to limit the size of the stack frame to 4 Kbytes."

As can be seen, the encoding of the x86 instruction set is very complex. This has to do partly with the need to be backward compatible with the 8086 machine and partly with a desire on the part of the designers to provide every possible assistance to the compiler writer in producing efficient code. It is a matter of some debate whether an instruction set as complex as this is preferable to the opposite extreme of the RISC instruction sets.

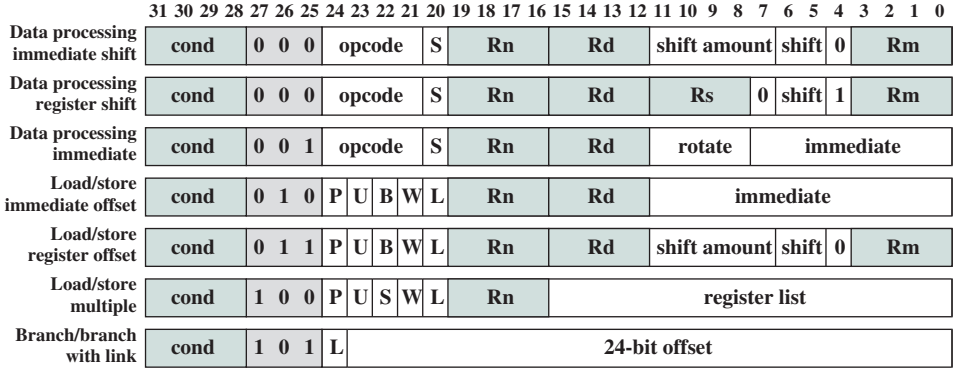
ARM Instruction Formats

All instructions in the ARM architecture are 32 bits long and follow a regular format (Figure 13.10). The first four bits of an instruction are the condition code. As discussed in Chapter 12, virtually all ARM instructions can be conditionally executed. The next three bits specify the general type of instruction. For most instructions other than branch instructions, the next five bits constitute an opcode and/or modifier bits for the operation. The remaining 20 bits are for operand addressing. The regular structure of the instruction formats eases the job of the instruction decode units.

IMMEDIATE CONSTANTS To achieve a greater range of immediate values, the data processing immediate format specifies both an immediate value and a rotate value. The 8-bit immediate value is expanded to 32 bits and then rotated right by a number of bits equal to twice the 4-bit rotate value. Several examples are shown in Figure 13.11.

THUMB INSTRUCTION SET The Thumb instruction set is a re-encoded subset of the ARM instruction set. Thumb is designed to increase the performance of ARM implementations that use a 16-bit or narrower memory data bus and to allow better code density than provided by the ARM instruction for both 16-bit and 32-bit processors. The Thumb instruction set was created by analyzing the 32-bit ARM instruction set and deriving the best fit 16-bit instruction set, thus reducing code size. The savings is achieved in the following way:

1. Thumb instructions are unconditional, so the condition code field is not used. Also, all Thumb arithmetic and logic instructions update the condition flags, so that the update-flag bit is not needed. Savings: 5 bits.



S = For data processing instructions, signifies that the instruction updates the condition codes

S = For load/store multiple instructions, signifies whether instruction execution is restricted to supervisor mode

P, U, W = bits that distinguish among different types of addressing mode

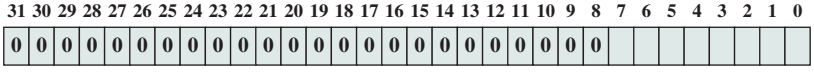
B = Distinguishes between an unsigned byte (B==1) and a word (B==0) access

L = For load/store instructions, distinguishes between a Load (L==1) and a Store (L==0)

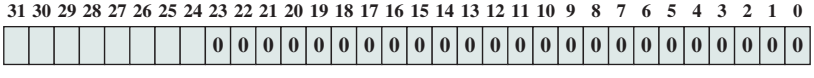
L = For branch instructions, determines whether a return address is stored in the link register

Figure 13.10 ARM Instruction Formats

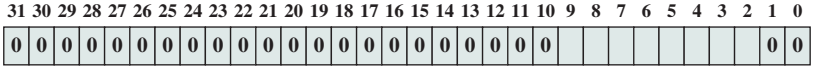
2. Thumb has only a subset of the operations in the full instruction set and uses only a 2-bit opcode field, plus a 3-bit type field. Savings: 2 bits.
3. The remaining savings of 9 bits comes from reductions in the operand specifications. For example, Thumb instructions reference only registers r0 through r7, so only 3 bits are required for register references, rather than 4 bits. Immediate values do not include a 4-bit rotate field.



ror #0—range 0 through 0x000000FF—step 0x00000001



ror #8—range 0 through 0xFF000000—step 0x01000000



ror #30—range 0 through 0x000003FC—step 0x00000004

Figure 13.11 Examples of Use of ARM Immediate Constants

The ARM processor can execute a program consisting of a mixture of Thumb instructions and 32-bit ARM instructions. A bit in the processor control register determines which type of instruction is currently being executed. Figure 13.12 shows an example. The figure shows both the general format and a specific instance of an instruction in both 16-bit and 32-bit formats.

THUMB-2 INSTRUCTION SET With the introduction of the Thumb instruction set, the user was required to blend instruction sets by compiling performance critical code to ARM and the rest to Thumb. This manual code blending requires additional effort and it is difficult to achieve optimal results. To overcome these problems, ARM developed the Thumb-2 instruction set, which is the only instruction set available on the Cortex-M microcontroller products.

Thumb-2 is a major enhancement to the Thumb instruction set architecture (ISA). It introduces 32-bit instructions that can be intermixed freely with the older 16-bit Thumb instructions. These new 32-bit instructions cover almost all the functionality of the ARM instruction set. The most important difference between the Thumb ISA and the ARM ISA is that most 32-bit Thumb instructions are unconditional, whereas almost all ARM instructions can be conditional. However, Thumb-2 introduces a new If-Then (IT) instruction that delivers much of the functionality of the condition field in ARM instructions. Thumb-2 delivers overall code density comparable with Thumb, together with the performance levels associated with the ARM ISA. Before Thumb-2, developers had to choose between Thumb for size and ARM for performance.

[ROBI07] reports on an analysis of the Thumb-2 instruction set compared with the ARM and original Thumb instruction sets. The analysis involved compiling and executing the Embedded Microprocessor Benchmark Consortium (EEMBC) benchmark suite using the three instruction sets, with the following results:

- With compilers optimized for performance, Thumb-2 size was 26% smaller than ARM, and slightly larger than original Thumb.
- With compilers optimized for space, Thumb-2 size was 32% smaller than ARM, and slightly smaller than original Thumb.

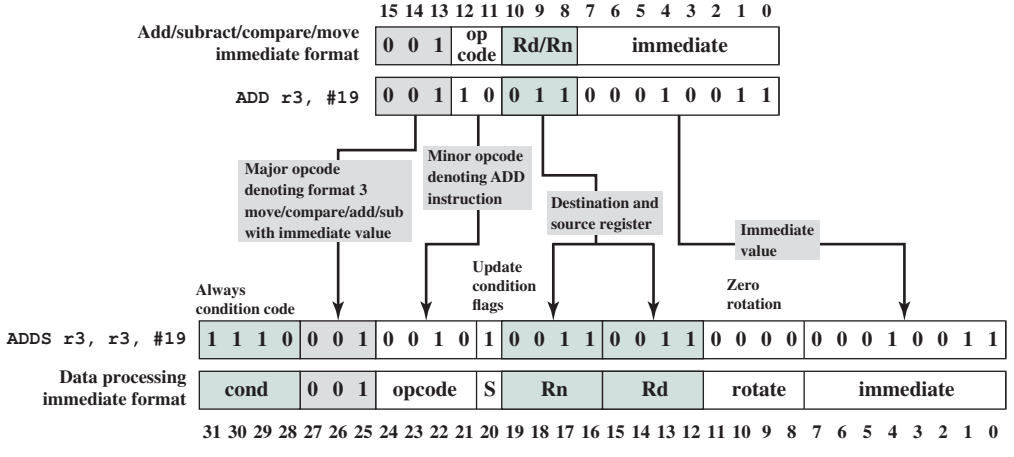
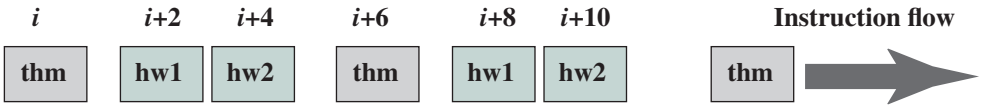

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Add/subract/compare/move immediate format
0 0 1
op code
Rd/Rn
immediate
ADD r3, #19
0 0 1 1 0 0 1 1 0 0 0 1 0 0 1 1
Major opcode denoting format 3 move/compare/add/sub with immediate value
Minor opcode denoting ADD instruction
Destination and source register
Immediate value
Always condition code
Update condition flags
Zero rotation
ADDS r3, r3, #19
1 1 1 0 0 0 1 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 0 0 0 1 0 0 1 1
Data processing immediate format
cond
0 0 1
opcode
S
Rn
Rd
rotate
immediate
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Figure 13.12 Expanding a Thumb ADD Instruction into its ARM Equivalent



Halfword1 [15:13]	Halfword1 [12:11]	Length	Functionality
Not 111	xx	16 bits (1 halfword)	16-bit Thumb instruction
111	00	16 bits (1 halfword)	16-bit Thumb unconditional branch instruction
111	Not 00	32 bits (2 halfwords)	32-bit Thumb-2 instruction

Figure 13.13 Thumb-2 Encoding

- With compilers optimized for performance, Thumb-2 performance on the benchmark suite was 98% of ARM performance and 125% of original Thumb performance.

These results confirm that Thumb-2 meets its design objectives.

Figure 13.13 shows how the new 32-bit Thumb instructions are encoded. The encoding is compatible with the existing Thumb unconditional branch instructions, which has the bit pattern 11100 in the five leftmost bits of the instruction. No other 16-bit instruction begins with the pattern 111 in the three leftmost bits, so the bit patterns 11101, 11110, and 11111 indicate that this is a 32-bit Thumb instruction.

13.5 ASSEMBLY LANGUAGE

A processor can understand and execute machine instructions. Such instructions are simply binary numbers stored in the computer. If a programmer wished to program directly in machine language, then it would be necessary to enter the program as binary data.

Consider the simple BASIC statement

$$N = I + J + K$$

Suppose we wished to program this statement in machine language and to initialize I, J, and K to 2, 3, and 4, respectively. This is shown in Figure 13.14a. The program starts in location 101 (hexadecimal). Memory is reserved for the four variables starting at location 201. The program consists of four instructions:

1. Load the contents of location 201 into the AC.
2. Add the contents of location 202 to the AC.
3. Add the contents of location 203 to the AC.
4. Store the contents of the AC in location 204.

This is clearly a tedious and very error-prone process.

A slight improvement is to write the program in hexadecimal rather than binary notation (Figure 13.14b). We could write the program as a series of lines. Each

Address	Contents			
101	0010	0010	101	2201
102	0001	0010	102	1202
103	0001	0010	103	1203
104	0011	0010	104	3204
201	0000	0000	201	0002
202	0000	0000	202	0003
203	0000	0000	203	0004
204	0000	0000	204	0000

(a) Binary program

Address	Contents
101	2201
102	1202
103	1203
104	3204
201	0002
202	0003
203	0004
204	0000

(b) Hexadecimal program

Address	Instruction	
101	LDA	201
102	ADD	202
103	ADD	203
104	STA	204
201	DAT	2
202	DAT	3
203	DAT	4
204	DAT	0

(c) Symbolic program

Label	Operation	Operand
FORMUL	LDA	I
	ADD	J
	ADD	K
	STA	N
I	DATA	2
J	DATA	3
K	DATA	4
N	DATA	0

(d) Assembly program

Figure 13.14 Computation of the Formula N = I + J + K

line contains the address of a memory location and the hexadecimal code of the binary value to be stored in that location. Then we need a program that will accept this input, translate each line into a binary number, and store it in the specified location.

For more improvement, we can make use of the symbolic name or mnemonic of each instruction. This results in the *symbolic program* shown in Figure 13.14c. Each line of input still represents one memory location. Each line consists of three fields, separated by spaces. The first field contains the address of a location. For an instruction, the second field contains the three-letter symbol for the opcode. If it is a memory-referencing instruction, then a third field contains the address. To store arbitrary data in a location, we invent a *pseudoinstruction* with the symbol DAT. This is merely an indication that the third field on the line contains a hexadecimal number to be stored in the location specified in the first field.

For this type of input we need a slightly more complex program. The program accepts each line of input, generates a binary number based on the second and third (if present) fields, and stores it in the location specified by the first field.

The use of a symbolic program makes life much easier but is still awkward. In particular, we must give an absolute address for each word. This means that the program and data can be loaded into only one place in memory, and we must know that place ahead of time. Worse, suppose we wish to change the program some day by adding or deleting a line. This will change the addresses of all subsequent words.

A much better system, and one commonly used, is to use symbolic addresses. This is illustrated in Figure 13.14d. Each line still consists of three fields. The first field is still for the address, but a symbol is used instead of an absolute numerical address. Some lines have no address, implying that the address of that line is one

more than the address of the previous line. For memory-reference instructions, the third field also contains a symbolic address.

With this last refinement, we have an *assembly language.* Programs written in assembly language (assembly programs) are translated into machine language by an *assembler.* This program must not only do the symbolic translation discussed earlier but also assign some form of memory addresses to symbolic addresses.

The development of assembly language was a major milestone in the evolution of computer technology. It was the first step to the high-level languages in use today. Although few programmers use assembly language, virtually all machines provide one. They are used, if at all, for systems programs such as compilers and I/O routines.

Appendix B provides a more detailed examination of assembly language.

13.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

autoindexing	immediate addressing	preindexing
base-register addressing	indexing	register addressing
direct addressing	indirect addressing	register indirect addressing
displacement addressing	instruction format	relative addressing
effective address	postindexing	word

Review Questions

13.1 Briefly define immediate addressing.
13.2 Briefly define **direct addressing**.
13.3 Briefly define indirect addressing.
13.4 Briefly define register addressing.
13.5 Briefly define register indirect addressing.
13.6 Briefly define displacement addressing.
13.7 Briefly define relative addressing.
13.8 What is the advantage of autoindexing?
13.9 What is the difference between postindexing and preindexing?
13.10 What facts go into determining the use of the addressing bits of an instruction?
13.11 What are the advantages and disadvantages of using a variable-length instruction format?

Problems

13.1 Given the following memory values and a one-address machine with an accumulator, what values do the following instructions load into the accumulator?
- Word 20 contains 40.
- Word 30 contains 50.
- Word 40 contains 60.
- Word 50 contains 70.

a. LOAD IMMEDIATE 20
b. LOAD DIRECT 20
c. LOAD INDIRECT 20
d. LOAD IMMEDIATE 30
e. LOAD DIRECT 30
f. LOAD INDIRECT 30

13.2 Let the address stored in the program counter be designated by the symbol X1. The instruction stored in X1 has an address part (operand reference) X2. The operand needed to execute the instruction is stored in the memory word with address X3. An index register contains the value X4. What is the relationship between these various quantities if the addressing mode of the instruction is (a) direct; (b) indirect; (c) PC relative; (d) indexed?

13.3 An address field in an instruction contains decimal value 14. Where is the corresponding operand located for
a. immediate addressing?
b. direct addressing?
c. indirect addressing?
d. register addressing?
e. register indirect addressing?

13.4 Consider a 16-bit processor in which the following appears in main memory, starting at location 200:

200	Load to AC	Mode
201	500	
202	Next instruction	

The first part of the first word indicates that this instruction loads a value into an accumulator. The Mode field specifies an addressing mode and, if appropriate, indicates a source register; assume that when used, the source register is R1, which has a value of 400. There is also a base register that contains the value 100. The value of 500 in location 201 may be part of the address calculation. Assume that location 399 contains the value 999, location 400 contains the value 1000, and so on. Determine the effective address and the operand to be loaded for the following address modes:

a. Direct
b. Immediate
c. Indirect
d. PC relative
e. Displacement
f. Register
g. Register indirect
h. Autoindexing with increment, using R1

13.5 A PC-relative mode branch instruction is 3 bytes long. The address of the instruction, in decimal, is 256028. Determine the branch target address if the signed displacement in the instruction is −31.

13.6 A PC-relative mode branch instruction is stored in memory at address 620_{10}. The branch is made to location 530_{10}. The address field in the instruction is 10 bits long. What is the binary value in the instruction?

13.7 How many times does the processor need to refer to memory when it fetches and executes an indirect-address-mode instruction if the instruction is (a) a computation requiring a single operand; (b) a branch?

13.8 The IBM 370 does not provide indirect addressing. Assume that the address of an operand is in main memory. How would you access the operand?

13.9 In [COOK82], the author proposes that the PC-relative addressing modes be eliminated in favor of other modes, such as the use of a stack. What is the disadvantage of this proposal?

13.10 The x86 includes the following instruction:

IMUL op1, op2, immediate

This instruction multiplies op2, which may be either register or memory, by the immediate operand value, and places the result in op1, which must be a register. There is no other three-operand instruction of this sort in the instruction set. What is the possible use of such an instruction? (*Hint*: Consider indexing.)

13.11 Consider a processor that includes a base with indexing addressing mode. Suppose an instruction is encountered that employs this addressing mode and specifies a displacement of 1970, in decimal. Currently the base and index register contain the decimal numbers 48,022 and 8, respectively. What is the address of the operand?

13.12 Define: EA = (X)+ is the effective address equal to the contents of location X, with X incremented by one word length after the effective address is calculated; EA = −(X) is the effective address equal to the contents of location X, with X decremented by one word length before the effective address is calculated; EA = (X)− is the effective address equal to the contents of location X, with X decremented by one word length after the effective address is calculated. Consider the following instructions, each in the format (Operation Source Operand, Destination Operand), with the result of the operation placed in the destination operand.

a. OP X, (X)
b. OP (X), (X)+
c. OP (X)+, (X)
d. OP − (X), (X)
e. OP − (X), (X)+
f. OP (X)+, (X)+
g. OP(X)−, (X)

Using X as the stack pointer, which of these instructions can pop the top two elements from the stack, perform the designated operation (e.g., ADD source to destination and store in destination), and push the result back on the stack? For each such instruction, does the stack grow toward memory location 0 or in the opposite direction?

13.13 Assume a stack-oriented processor that includes the stack operations PUSH and POP. Arithmetic operations automatically involve the top one or two stack elements. Begin with an empty stack. What stack elements remain after the following instructions are executed?

```
PUSH    4
PUSH    7
PUSH    8
ADD
PUSH    10
SUB
MUL
```

13.14 Justify the assertion that a 32-bit instruction is probably much less than twice as useful as a 16-bit instruction.

13.15 Why was IBM's decision to move from 36 bits to 32 bits per word wrenching, and to whom?

13.16 Assume an instruction set that uses a fixed 16-bit instruction length. Operand specifiers are 6 bits in length. There are K two-operand instructions and L zero-operand instructions. What is the maximum number of one-operand instructions that can be supported?

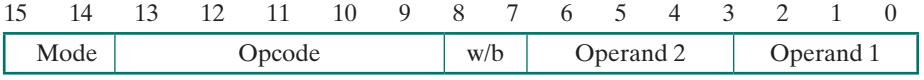
13.17 Design a variable-length opcode to allow all of the following to be encoded in a 36-bit instruction:

- instructions with two 15-bit addresses and one 3-bit register number;
- instructions with one 15-bit address and one 3-bit register number;
- instructions with no addresses or registers.

13.18 Consider the results of Problem 10.6. Assume that M is a 16-bit memory address and that X, Y, and Z are either 16-bit addresses or 4-bit register numbers. The one-address machine uses an accumulator, and the two- and three-address machines have 16 registers and instructions operating on all combinations of memory locations and registers. Assuming 8-bit opcodes and instruction lengths that are multiples of 4 bits, how many bits does each machine need to compute X?

13.19 Is there any possible justification for an instruction with two opcodes?

13.20 The 16-bit Zilog Z8001 has the following general instruction format:



The *mode* field specifies how to locate the operands from the *operand* fields. The *w/b* field is used in certain instructions to specify whether the operands are bytes or 16-bit words. The *operand 1* field may (depending on the *mode field* contents) specify one of 16 general-purpose registers. The *operand 2* field may specify any general-purpose registers except register 0. When the *operand 2* field is all zeros, each of the original opcodes takes on a new meaning.

a. How many opcodes are provided on the Z8001?

b. Suggest an efficient way to provide more opcodes and indicate the trade-off involved.

CHAPTER 14

Processor Structure and Function

14.1 Processor Organization

14.2 Register Organization
- User-Visible Registers
- Control and Status Registers
- Example Microprocessor Register Organizations

14.3 Instruction Cycle
- The Indirect Cycle
- Data Flow

14.4 Instruction Pipelining
- Pipelining Strategy
- Pipeline Performance
- Pipeline Hazards
- Dealing with Branches
- Intel 80486 Pipelining

14.5 The x86 Processor Family
- Register Organization
- Interrupt Processing

14.6 The Arm Processor
- Processor Organization
- Processor Modes
- Register Organization
- Interrupt Processing

14.7 Key Terms, Review Questions, and Problems

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Distinguish between **user-visible** and **control/status registers**, and discuss the purposes of registers in each category.
- Summarize the **instruction cycle**.
- Discuss the principle behind **instruction pipelining** and how it works in practice.
- Compare and contrast the various forms of pipeline hazards.
- Present an overview of the x86 processor structure.
- Present an overview of the ARM processor structure.

This chapter discusses aspects of the processor not yet covered in Part Three and sets the stage for the discussion of RISC and superscalar architecture in Chapters 15 and 16.

We begin with a summary of processor organization. Registers, which form the internal memory of the processor, are then analyzed. We are then in a position to return to the discussion (begun in Section 3.2) of the instruction cycle. A description of the instruction cycle and a common technique known as instruction pipelining complete our description. The chapter concludes with an examination of some aspects of the x86 and ARM organizations.

14.1 PROCESSOR ORGANIZATION

To understand the organization of the processor, let us consider the requirements placed on the processor, the things that it must do:

- **Fetch instruction:** The processor reads an instruction from memory (register, cache, main memory).
- **Interpret instruction:** The instruction is decoded to determine what action is required.
- **Fetch data:** The execution of an instruction may require reading data from memory or an I/O module.
- **Process data:** The execution of an instruction may require performing some arithmetic or logical operation on data.
- **Write data:** The results of an execution may require writing data to memory or an I/O module.

To do these things, it should be clear that the processor needs to store some data temporarily. It must remember the location of the last instruction so that it can know where to get the next instruction. It needs to store instructions and data temporarily while an instruction is being executed. In other words, the processor needs a small internal memory.

Figure 14.1 is a simplified view of a processor, indicating its connection to the rest of the system via the system bus. A similar interface would be needed for any

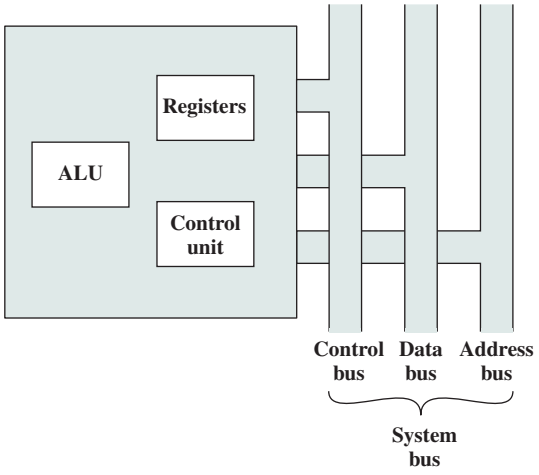


Figure 14.1 The CPU with the System Bus

of the interconnection structures described in Chapter 3. The reader will recall that the major components of the processor are an *arithmetic and logic unit* (ALU) and a *control unit* (CU). The ALU does the actual computation or processing of data. The control unit controls the movement of data and instructions into and out of the processor and controls the operation of the ALU. In addition, the figure shows a minimal internal memory, consisting of a set of storage locations, called *registers.*

Figure 14.2 is a slightly more detailed view of the processor. The data transfer and logic control paths are indicated, including an element labeled *internal*

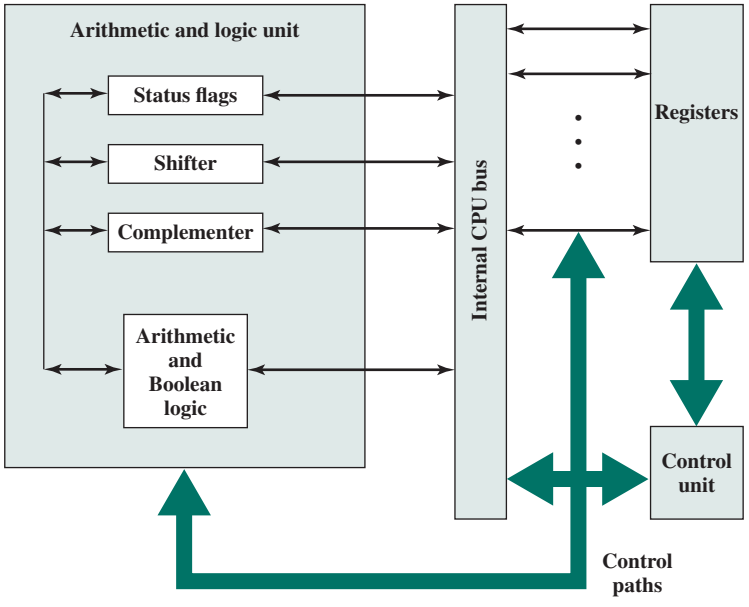


Figure 14.2 Internal Structure of the CPU

processor bus. This element is needed to transfer data between the various registers and the ALU because the ALU in fact operates only on data in the internal processor memory. The figure also shows typical basic elements of the ALU. Note the similarity between the internal structure of the computer as a whole and the internal structure of the processor. In both cases, there is a small collection of major elements (computer: processor, I/O, memory; processor: control unit, ALU, registers) connected by data paths.

14.2 REGISTER ORGANIZATION

As we discussed in Chapter 4, a computer system employs a memory hierarchy. At higher levels of the hierarchy, memory is faster, smaller, and more expensive (per bit). Within the processor, there is a set of registers that function as a level of memory above main memory and cache in the hierarchy. The registers in the processor perform two roles:

- **User-visible registers:** Enable the machine- or assembly language programmer to minimize main memory references by optimizing use of registers.
- **Control and status registers:** Used by the control unit to control the operation of the processor and by privileged, operating system programs to control the execution of programs.

There is not a clean separation of registers into these two categories. For example, on some machines the program counter is user visible (e.g., x86), but on many it is not. For purposes of the following discussion, however, we will use these categories.

User-Visible Registers

A user-visible register is one that may be referenced by means of the machine language that the processor executes. We can characterize these in the following categories:

- General purpose
- Data
- Address
- Condition codes

General-purpose registers can be assigned to a variety of functions by the programmer. Sometimes their use within the instruction set is orthogonal to the operation. That is, any general-purpose register can contain the operand for any opcode. This provides true general-purpose register use. Often, however, there are restrictions. For example, there may be dedicated registers for floating-point and stack operations.

In some cases, general-purpose registers can be used for addressing functions (e.g., register indirect, displacement). In other cases, there is a partial or clean separation between data registers and address registers. **Data registers** may be used only to hold data and cannot be employed in the calculation of an operand address.

Address registers may themselves be somewhat general purpose, or they may be devoted to a particular addressing mode. Examples include the following:

- **Segment pointers:** In a machine with segmented addressing (see Section 8.3), a segment register holds the address of the base of the segment. There may be multiple registers: for example, one for the operating system and one for the current process.
- **Index registers:** These are used for indexed addressing and may be autoindexed.
- **Stack pointer:** If there is user-visible stack addressing, then typically there is a dedicated register that points to the top of the stack. This allows implicit addressing; that is, push, pop, and other stack instructions need not contain an explicit stack operand.

There are several design issues to be addressed here. An important issue is whether to use completely general-purpose registers or to specialize their use. We have already touched on this issue in the preceding chapter because it affects instruction set design. With the use of specialized registers, it can generally be implicit in the opcode which type of register a certain operand specifier refers to. The operand specifier must only identify one of a set of specialized registers rather than one out of all the registers, thus saving bits. On the other hand, this specialization limits the programmer's flexibility.

Another design issue is the number of registers, either general purpose or data plus address, to be provided. Again, this affects instruction set design because more registers require more operand specifier bits. As we previously discussed, somewhere between 8 and 32 registers appears optimum [LUND77]. Fewer registers result in more memory references; more registers do not noticeably reduce memory references (e.g., see [WILL90]). However, a new approach, which finds advantage in the use of hundreds of registers, is exhibited in some RISC systems and is discussed in Chapter 15.

Finally, there is the issue of register length. Registers that must hold addresses obviously must be at least long enough to hold the largest address. Data registers should be able to hold values of most data types. Some machines allow two contiguous registers to be used as one for holding double-length values.

A final category of registers, which is at least partially visible to the user, holds **condition codes** (also referred to as *flags*). Condition codes are bits set by the processor hardware as the result of operations. For example, an arithmetic operation may produce a positive, negative, zero, or overflow result. In addition to the result itself being stored in a register or memory, a condition code is also set. The code may subsequently be tested as part of a conditional branch operation.

Condition code bits are collected into one or more registers. Usually, they form part of a control register. Generally, machine instructions allow these bits to be read by implicit reference, but the programmer cannot alter them.

Many processors, including those based on the IA-64 architecture and the MIPS processors, do not use condition codes at all. Rather, conditional branch instructions specify a comparison to be made and act on the result of the comparison, without storing a condition code. Table 14.1, based on [DERO87], lists key advantages and disadvantages of condition codes.

Table 14.1 Condition Codes

Advantages	Disadvantages
1. Because condition codes are set by normal arithmetic and data movement instructions, they should reduce the number of COMPARE and TEST instructions needed.	1. Condition codes add complexity, both to the hardware and software. Condition code bits are often modified in different ways by different instructions, making life more difficult for both the microprogrammer and compiler writer.
2. Conditional instructions, such as BRANCH are simplified relative to composite instructions, such as TEST and BRANCH.	2. Condition codes are irregular; they are typically not part of the main data path, so they require extra hardware connections.
3. Condition codes facilitate multiway branches. For example, a TEST instruction can be followed by two branches, one on less than or equal to zero and one on greater than zero.	3. Often condition code machines must add special non-condition-code instructions for special situations anyway, such as bit checking, loop control, and atomic semaphore operations.
4. Condition codes can be saved on the stack during subroutine calls along with other register information.	4. In a pipelined implementation, condition codes require special synchronization to avoid conflicts.

In some machines, a subroutine call will result in the automatic saving of all user-visible registers, to be restored on return. The processor performs the saving and restoring as part of the execution of call and return instructions. This allows each subroutine to use the user-visible registers independently. On other machines, it is the responsibility of the programmer to save the contents of the relevant user-visible registers prior to a subroutine call, by including instructions for this purpose in the program.

Control and Status Registers

There are a variety of processor registers that are employed to control the operation of the processor. Most of these, on most machines, are not visible to the user. Some of them may be visible to machine instructions executed in a control or operating system mode.

Of course, different machines will have different register organizations and use different terminology. We list here a reasonably complete list of register types, with a brief description.

Four registers are essential to instruction execution:

- **Program counter (PC):** Contains the address of an instruction to be fetched.
- **Instruction register (IR):** Contains the instruction most recently fetched.
- **Memory address register (MAR):** Contains the address of a location in memory.
- **Memory buffer register (MBR):** Contains a word of data to be written to memory or the word most recently read.

Not all processors have internal registers designated as MAR and MBR, but some equivalent buffering mechanism is needed whereby the bits to be transferred

to the system bus are staged and the bits to be read from the data bus are temporarily stored.

Typically, the processor updates the PC after each instruction fetch so that the PC always points to the next instruction to be executed. A branch or skip instruction will also modify the contents of the PC. The fetched instruction is loaded into an IR, where the opcode and operand specifiers are analyzed. Data are exchanged with memory using the MAR and MBR. In a bus-organized system, the MAR connects directly to the address bus, and the MBR connects directly to the data bus. User-visible registers, in turn, exchange data with the MBR.

The four registers just mentioned are used for the movement of data between the processor and memory. Within the processor, data must be presented to the ALU for processing. The ALU may have direct access to the MBR and user-visible registers. Alternatively, there may be additional buffering registers at the boundary to the ALU; these registers serve as input and output registers for the ALU and exchange data with the MBR and user-visible registers.

Many processor designs include a register or set of registers, often known as the *program status word* (PSW), that contain status information. The PSW typically contains condition codes plus other status information. Common fields or flags include the following:

- **Sign:** Contains the sign bit of the result of the last arithmetic operation.
- **Zero:** Set when the result is 0.
- **Carry:** Set if an operation resulted in a carry (addition) into or borrow (subtraction) out of a high-order bit. Used for multiword arithmetic operations.
- **Equal:** Set if a logical compare result is equality.
- **Overflow:** Used to indicate arithmetic overflow.
- **Interrupt Enable/Disable:** Used to enable or disable interrupts.
- **Supervisor:** Indicates whether the processor is executing in supervisor or user mode. Certain privileged instructions can be executed only in supervisor mode, and certain areas of memory can be accessed only in supervisor mode.

A number of other registers related to status and control might be found in a particular processor design. There may be a pointer to a block of memory containing additional status information (e.g., process control blocks). In machines using vectored interrupts, an interrupt vector register may be provided. If a stack is used to implement certain functions (e.g., subroutine call), then a system stack pointer is needed. A page table pointer is used with a virtual memory system. Finally, registers may be used in the control of I/O operations.

A number of factors go into the design of the control and status register organization. One key issue is operating system support. Certain types of control information are of specific utility to the operating system. If the processor designer has a functional understanding of the operating system to be used, then the register organization can to some extent be tailored to the operating system.

Another key design decision is the allocation of control information between registers and memory. It is common to dedicate the first (lowest) few hundred or

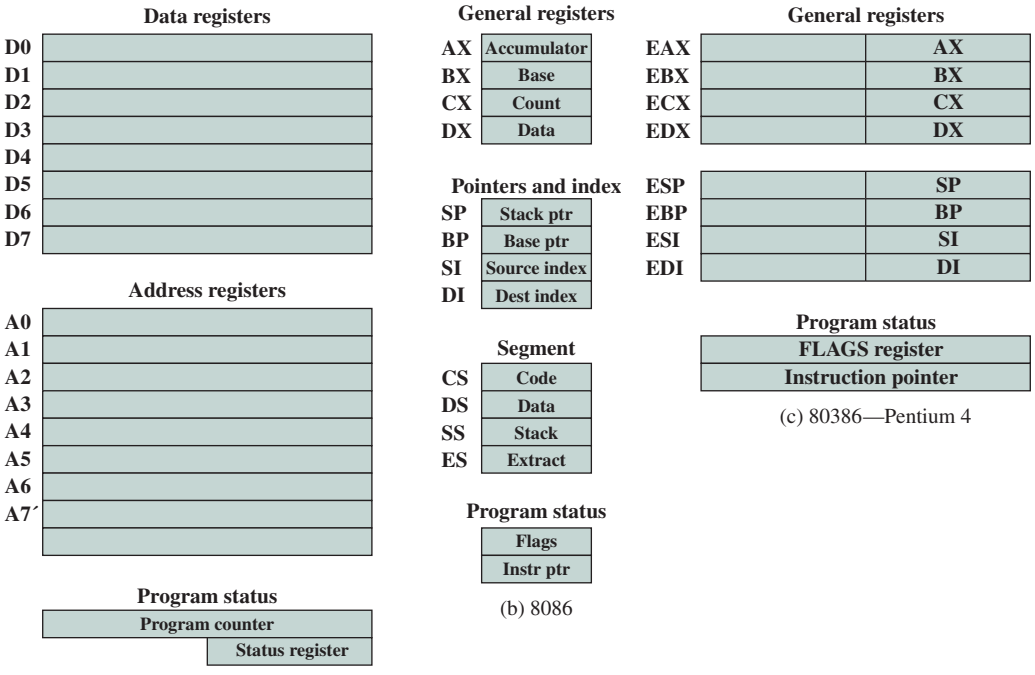


Figure 14.3 Example Microprocessor Register Organizations

thousand words of memory for control purposes. The designer must decide how much control information should be in registers and how much in memory. The usual trade-off of cost versus speed arises.

Example Microprocessor Register Organizations

It is instructive to examine and compare the register organization of comparable systems. In this section, we look at two 16-bit microprocessors that were designed at about the same time: the Motorola MC68000 [STRI79] and the Intel 8086 [MORS78]. Figures 14.3a and b depict the register organization of each; purely internal registers, such as a memory address register, are not shown.

The MC68000 partitions its 32-bit registers into eight data registers and nine address registers. The eight data registers are used primarily for data manipulation and are also used in addressing as index registers. The width of the registers allows 8-, 16-, and 32-bit data operations, determined by opcode. The address registers contain 32-bit (no segmentation) addresses; two of these registers are also used as stack pointers, one for users and one for the operating system, depending on the current execution mode. Both registers are numbered 7, because only one can be used at a time. The MC68000 also includes a 32-bit program counter and a 16-bit status register.

The Motorola team wanted a very regular instruction set, with no special-purpose registers. A concern for code efficiency led them to divide the registers into

two functional components, saving one bit on each register specifier. This seems a reasonable compromise between complete generality and code compaction.

The Intel 8086 takes a different approach to register organization. Every register is special purpose, although some registers are also usable as general purpose. The 8086 contains four 16-bit data registers that are addressable on a byte or 16-bit basis, and four 16-bit pointer and index registers. The data registers can be used as general purpose in some instructions. In others, the registers are used implicitly. For example, a multiply instruction always uses the accumulator. The four pointer registers are also used implicitly in a number of operations; each contains a segment offset. There are also four 16-bit segment registers. Three of the four segment registers are used in a dedicated, implicit fashion, to point to the segment of the current instruction (useful for branch instructions), a segment containing data, and a segment containing a stack, respectively. These dedicated and implicit uses provide for compact encoding at the cost of reduced flexibility. The 8086 also includes an instruction pointer and a set of 1-bit status and control flags.

The point of this comparison should be clear. There is no universally accepted philosophy concerning the best way to organize processor registers [TOON81]. As with overall instruction set design and so many other processor design issues, it is still a matter of judgment and taste.

A second instructive point concerning register organization design is illustrated in Figure 14.3c. This figure shows the user-visible register organization for the Intel 80386 [ELAY85], which is a 32-bit microprocessor designed as an extension of the 8086.[1] The 80386 uses 32-bit registers. However, to provide upward compatibility for programs written on the earlier machine, the 80386 retains the original register organization embedded in the new organization. Given this design constraint, the architects of the 32-bit processors had limited flexibility in designing the register organization.

14.3 INSTRUCTION CYCLE

In Section 3.2, we described the processor's instruction cycle (Figure 3.9). To recall, an instruction cycle includes the following stages:

- **Fetch:** Read the next instruction from memory into the processor.
- **Execute:** Interpret the opcode and perform the indicated operation.
- **Interrupt:** If interrupts are enabled and an interrupt has occurred, save the current process state and service the interrupt.

We are now in a position to elaborate somewhat on the instruction cycle. First, we must introduce one additional stage, known as the indirect cycle.

[1]Because the MC68000 already uses 32-bit registers, the MC68020 [MACD84], which is a full 32-bit architecture, uses the same register organization.

The Indirect Cycle

We have seen, in Chapter 13, that the execution of an instruction may involve one or more operands in memory, each of which requires a memory access. Further, if indirect addressing is used, then additional memory accesses are required.

We can think of the fetching of indirect addresses as one more instruction stages. The result is shown in Figure 14.4. The main line of activity consists of alternating instruction fetch and instruction execution activities. After an instruction is fetched, it is examined to determine if any indirect addressing is involved. If so, the required operands are fetched using indirect addressing. Following execution, an interrupt may be processed before the next instruction fetch.

Another way to view this process is shown in Figure 14.5, which is a revised version of Figure 3.12. This illustrates more correctly the nature of the instruction cycle. Once an instruction is fetched, its operand specifiers must be identified. Each input operand in memory is then fetched, and this process may require indirect addressing. Register-based operands need not be fetched. Once the opcode is executed, a similar process may be needed to store the result in main memory.

Data Flow

The exact sequence of events during an instruction cycle depends on the design of the processor. We can, however, indicate in general terms what must happen. Let us assume that a processor that employs a memory address register (MAR), a memory buffer register (MBR), a program counter (PC), and an instruction register (IR).

During the *fetch cycle,* an instruction is read from memory. Figure 14.6 shows the flow of data during this cycle. The PC contains the address of the next instruction to be fetched. This address is moved to the MAR and placed on the address bus. The control unit requests a memory read, and the result is placed on the data bus and copied into the MBR and then moved to the IR. Meanwhile, the PC is incremented by 1, preparatory for the next fetch.

Once the fetch cycle is over, the control unit examines the contents of the IR to determine if it contains an operand specifier using indirect addressing. If so, an

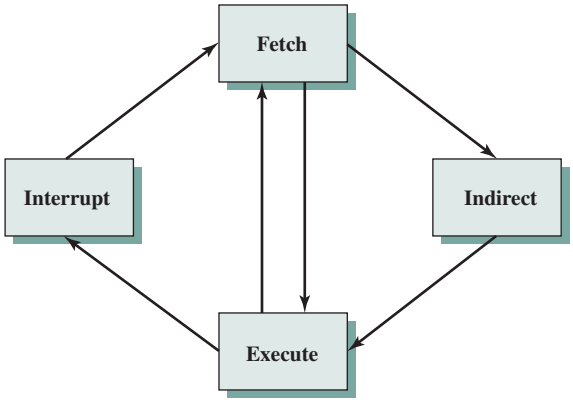


Figure 14.4 The Instruction Cycle

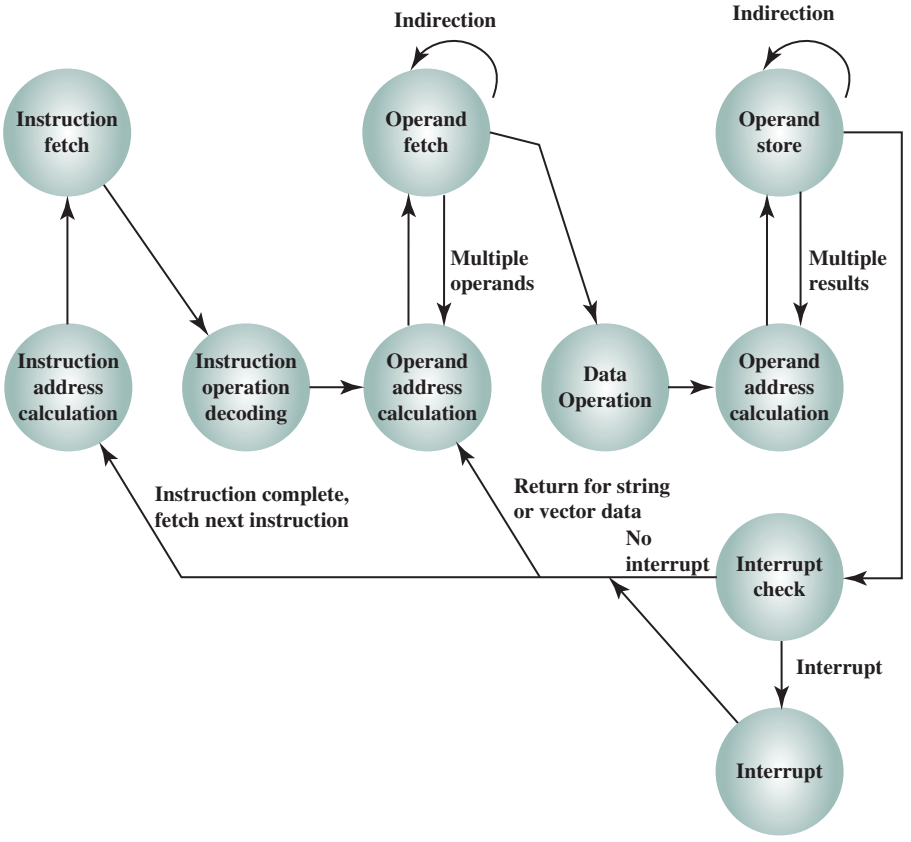
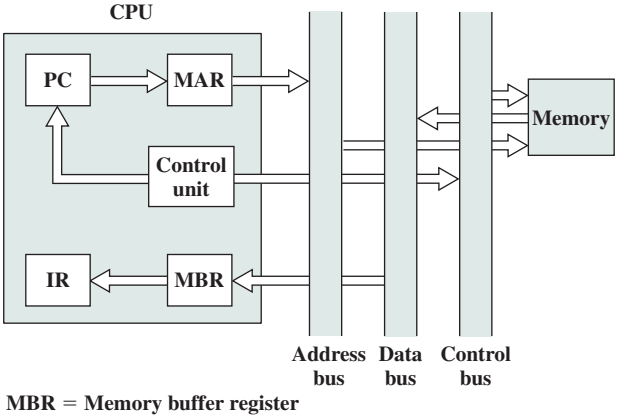


Figure 14.5 Instruction Cycle State Diagram



MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

Figure 14.6 Data Flow, Fetch Cycle

indirect cycle is performed. As shown in Figure 14.7, this is a simple cycle. The rightmost *N* bits of the MBR, which contain the address reference, are transferred to the MAR. Then the control unit requests a memory read, to get the desired address of the operand into the MBR.

The fetch and indirect cycles are simple and predictable. The *execute cycle* takes many forms; the form depends on which of the various machine instructions is in the IR. This cycle may involve transferring data among registers, read or write from memory or I/O, and/or the invocation of the ALU.

Like the fetch and indirect cycles, the *interrupt cycle* is simple and predictable (Figure 14.8). The current contents of the PC must be saved so that the processor can resume normal activity after the interrupt. Thus, the contents of the PC are transferred to the MBR to be written into memory. The special memory location reserved for this purpose is loaded into the MAR from the control unit. It might, for example, be a stack pointer. The PC is loaded with the address of the interrupt routine. As a result, the next instruction cycle will begin by fetching the appropriate instruction.

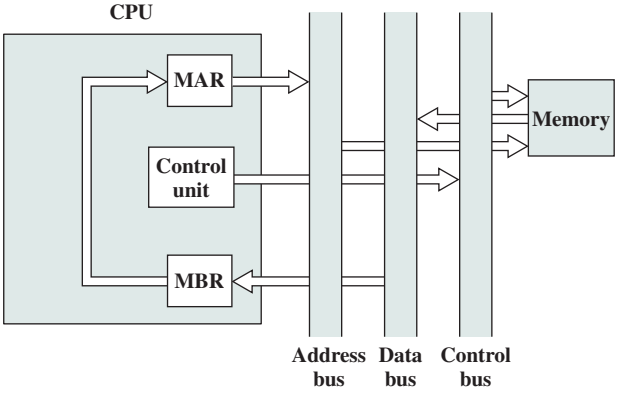


Figure 14.7 Data Flow, Indirect Cycle

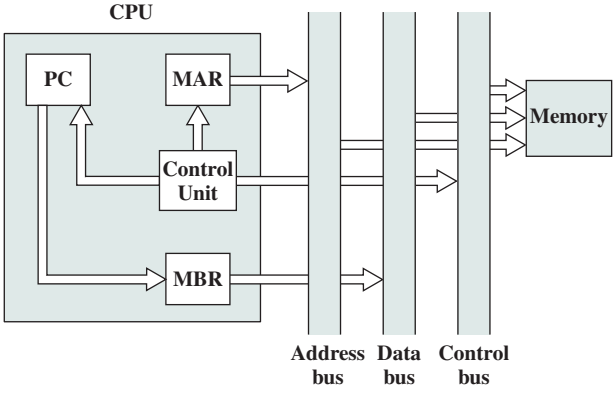


Figure 14.8 Data Flow, Interrupt Cycle

14.4 INSTRUCTION PIPELINING

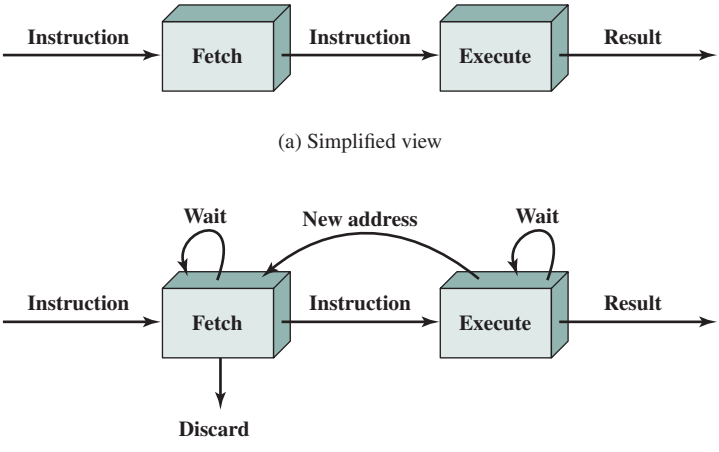
As computer systems evolve, greater performance can be achieved by taking advantage of improvements in technology, such as faster circuitry. In addition, organizational enhancements to the processor can improve performance. We have already seen some examples of this, such as the use of multiple registers rather than a single accumulator, and the use of a cache memory. Another organizational approach, which is quite common, is instruction pipelining.

Pipelining Strategy

Instruction pipelining is similar to the use of an assembly line in a manufacturing plant. An assembly line takes advantage of the fact that a product goes through various stages of production. By laying the production process out in an assembly line, products at various stages can be worked on simultaneously. This process is also referred to as *pipelining,* because, as in a pipeline, new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.

To apply this concept to instruction execution, we must recognize that, in fact, an instruction has a number of stages. Figures 14.5, for example, breaks the instruction cycle up into 10 tasks, which occur in sequence. Clearly, there should be some opportunity for pipelining.

As a simple approach, consider subdividing instruction processing into two stages: fetch instruction and execute instruction. There are times during the execution of an instruction when main memory is not being accessed. This time could be used to fetch the next instruction in parallel with the execution of the current one. Figure 14.9a depicts this approach. The pipeline has two independent stages. The first stage fetches an instruction and buffers it. When the second stage is free, the first stage passes it the buffered instruction. While the second stage is executing the instruction, the first stage takes advantage of any unused memory cycles to fetch



(a) Simplified view

(b) Expanded view

Figure 14.9 Two-Stage Instruction Pipeline

and buffer the next instruction. This is called instruction prefetch or *fetch overlap.* Note that this approach, which involves instruction buffering, requires more registers. In general, pipelining requires registers to store data between stages.

It should be clear that this process will speed up instruction execution. If the fetch and execute stages were of equal duration, the instruction cycle time would be halved. However, if we look more closely at this pipeline (Figure 14.9b), we will see that this doubling of execution rate is unlikely for two reasons:

1. The execution time will generally be longer than the fetch time. Execution will involve reading and storing operands and the performance of some operation. Thus, the fetch stage may have to wait for some time before it can empty its buffer.
2. A conditional branch instruction makes the address of the next instruction to be fetched unknown. Thus, the fetch stage must wait until it receives the next instruction address from the execute stage. The execute stage may then have to wait while the next instruction is fetched.

Guessing can reduce the time loss from the second reason. A simple rule is the following: When a conditional branch instruction is passed on from the fetch to the execute stage, the fetch stage fetches the next instruction in memory after the branch instruction. Then, if the branch is not taken, no time is lost. If the branch is taken, the fetched instruction must be discarded and a new instruction fetched.

While these factors reduce the potential effectiveness of the two-stage pipeline, some speedup occurs. To gain further speedup, the pipeline must have more stages. Let us consider the following decomposition of the instruction processing.

- **Fetch instruction (FI):** Read the next expected instruction into a buffer.
- **Decode instruction (DI):** Determine the opcode and the operand specifiers.
- **Calculate operands (CO):** Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect, or other forms of address calculation.
- **Fetch operands (FO):** Fetch each operand from memory. Operands in registers need not be fetched.
- **Execute instruction (EI):** Perform the indicated operation and store the result, if any, in the specified destination operand location.
- **Write operand (WO):** Store the result in memory.

With this decomposition, the various stages will be of more nearly equal duration. For the sake of illustration, let us assume equal duration. Using this assumption, Figure 14.10 shows that a six-stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.

Several comments are in order: The diagram assumes that each instruction goes through all six stages of the pipeline. This will not always be the case. For example, a load instruction does not need the WO stage. However, to simplify the pipeline hardware, the timing is set up assuming that each instruction requires all six stages. Also, the diagram assumes that all of the stages can be performed in parallel. In particular, it is assumed that there are no memory conflicts. For example, the FI, FO, and WO stages involve a memory access. The diagram implies that all these accesses can occur simultaneously. Most memory systems will not permit that.

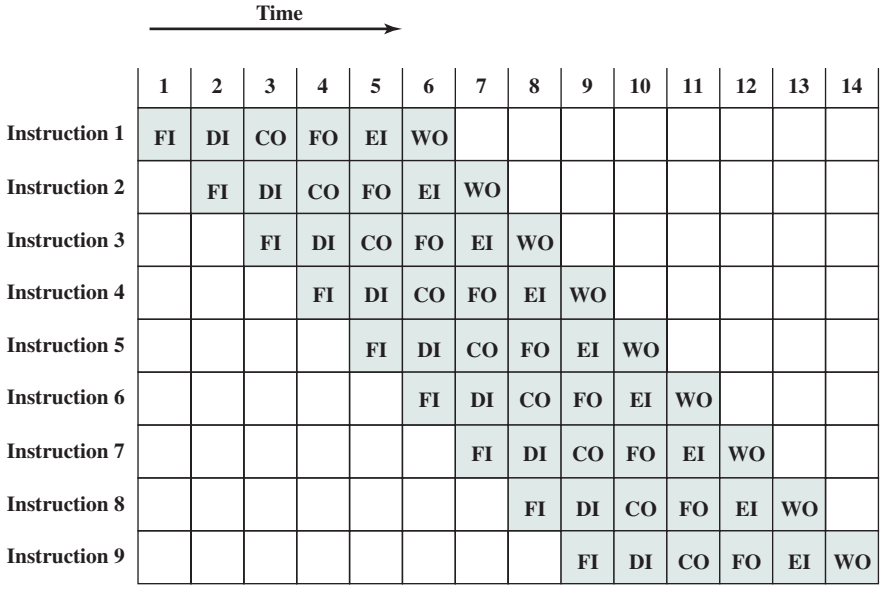

Time
1 2 3 4 5 6 7 8 9 10 11 12 13 14
Instruction 1 FI DI CO FO EI WO
Instruction 2 FI DI CO FO EI WO
Instruction 3 FI DI CO FO EI WO
Instruction 4 FI DI CO FO EI WO
Instruction 5 FI DI CO FO EI WO
Instruction 6 FI DI CO FO EI WO
Instruction 7 FI DI CO FO EI WO
Instruction 8 FI DI CO FO EI WO
Instruction 9 FI DI CO FO EI WO

Figure 14.10 Timing Diagram for Instruction Pipeline Operation

However, the desired value may be in cache, or the FO or WO stage may be null. Thus, much of the time, memory conflicts will not slow down the pipeline.

Several other factors serve to limit the performance enhancement. If the six stages are not of equal duration, there will be some waiting involved at various pipeline stages, as discussed before for the two-stage pipeline. Another difficulty is the conditional branch instruction, which can invalidate several instruction fetches. A similar unpredictable event is an interrupt. Figure 14.11 illustrates the effects of the conditional branch, using the same program as Figure 14.10. Assume that instruction 3 is a conditional branch to instruction 15. Until the instruction is executed, there is no way of knowing which instruction will come next. The pipeline, in this example, simply loads the next instruction in sequence (instruction 4) and proceeds. In Figure 14.10, the branch is not taken, and we get the full performance benefit of the enhancement. In Figure 14.11, the branch is taken. This is not determined until the end of time unit 7. At this point, the pipeline must be cleared of instructions that are not useful. During time unit 8, instruction 15 enters the pipeline. No instructions complete during time units 9 through 12; this is the performance penalty incurred because we could not anticipate the branch. Figure 14.12 indicates the logic needed for pipelining to account for branches and interrupts.

Other problems arise that did not appear in our simple two-stage organization. The CO stage may depend on the contents of a register that could be altered by a previous instruction that is still in the pipeline. Other such register and memory conflicts could occur. The system must contain logic to account for this type of conflict.

To clarify pipeline operation, it might be useful to look at an alternative depiction. Figures 14.10 and 14.11 show the progression of time horizontally across the figures, with each row showing the progress of an individual instruction. Figure 14.13 shows same sequence of events, with time progressing vertically down

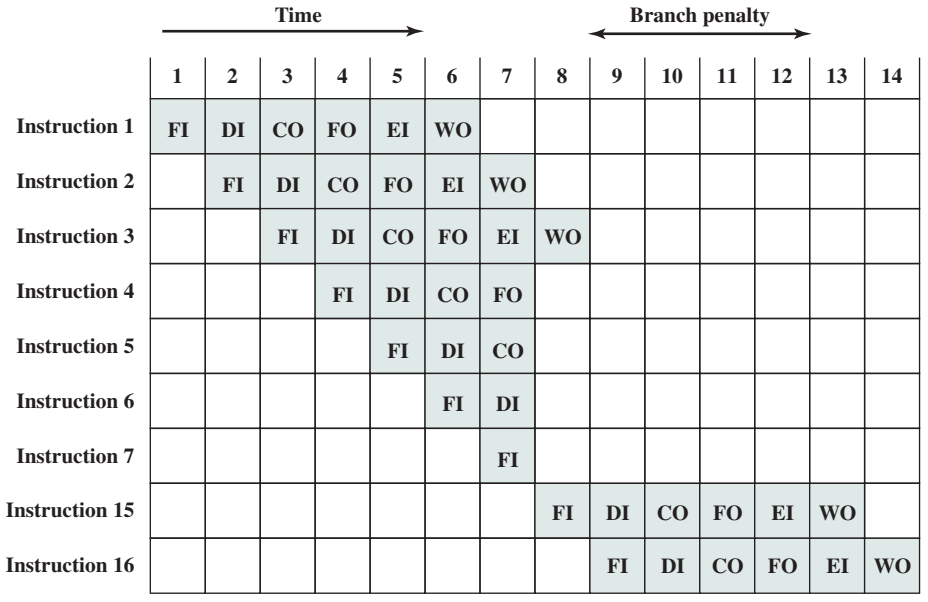

Time
Branch penalty
1 2 3 4 5 6 7 8 9 10 11 12 13 14
Instruction 1 FI DI CO FO EI WO
Instruction 2 FI DI CO FO EI WO
Instruction 3 FI DI CO FO EI WO
Instruction 4 FI DI CO FO
Instruction 5 FI DI CO
Instruction 6 FI DI
Instruction 7 FI
Instruction 15 FI DI CO FO EI WO
Instruction 16 FI DI CO FO EI WO

Figure 14.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

the figure, and each row showing the state of the pipeline at a given point in time. In Figure 14.13a (which corresponds to Figure 14.10), the pipeline is full at time 6, with 6 different instructions in various stages of execution, and remains full through time 9; we assume that instruction I9 is the last instruction to be executed. In Figure 14.13b, (which corresponds to Figure 14.11), the pipeline is full at times 6 and 7. At time 7, instruction 3 is in the execute stage and executes a branch to instruction 15. At this point, instructions I4 through I7 are flushed from the pipeline, so that at time 8, only two instructions are in the pipeline, I3 and I15.

From the preceding discussion, it might appear that the greater the number of stages in the pipeline, the faster the execution rate. Some of the IBM S/360 designers pointed out two factors that frustrate this seemingly simple pattern for high-performance design [ANDE67a], and they remain elements that designer must still consider:

1. At each stage of the pipeline, there is some overhead involved in moving data from buffer to buffer and in performing various preparation and delivery functions. This overhead can appreciably lengthen the total execution time of a single instruction. This is significant when sequential instructions are logically dependent, either through heavy use of branching or through memory access dependencies.

2. The amount of control logic required to handle memory and register dependencies and to optimize the use of the pipeline increases enormously with the number of stages. This can lead to a situation where the logic controlling the gating between stages is more complex than the stages being controlled.

Another consideration is latching delay: It takes time for pipeline buffers to operate and this adds to instruction cycle time.

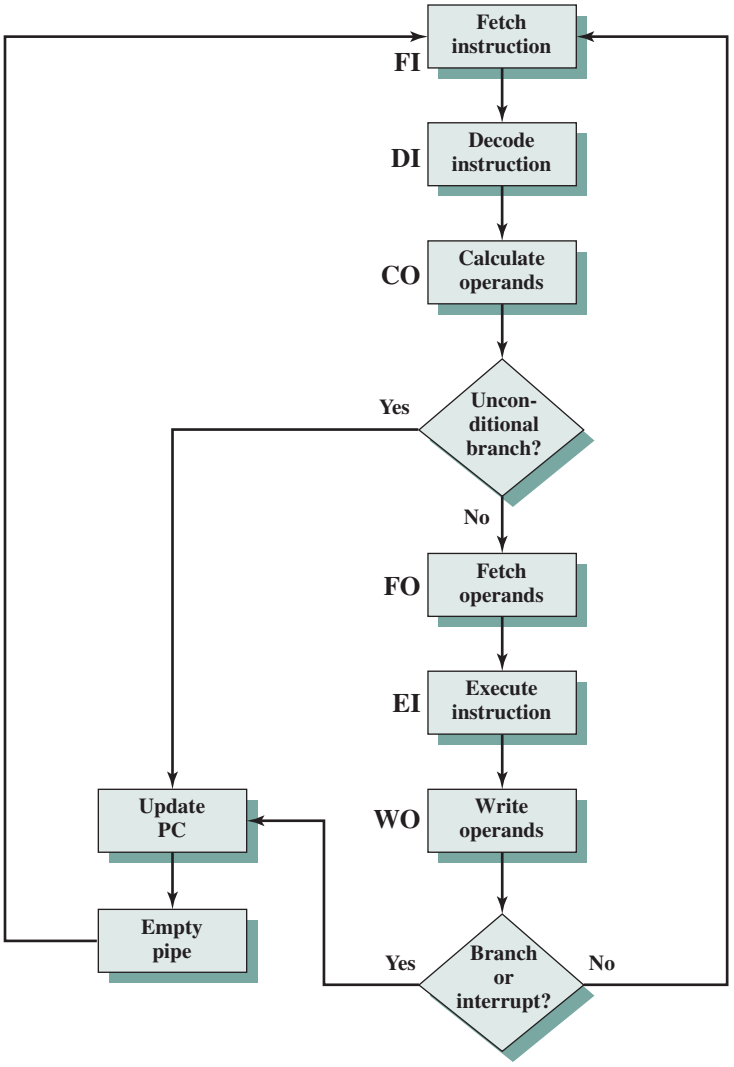


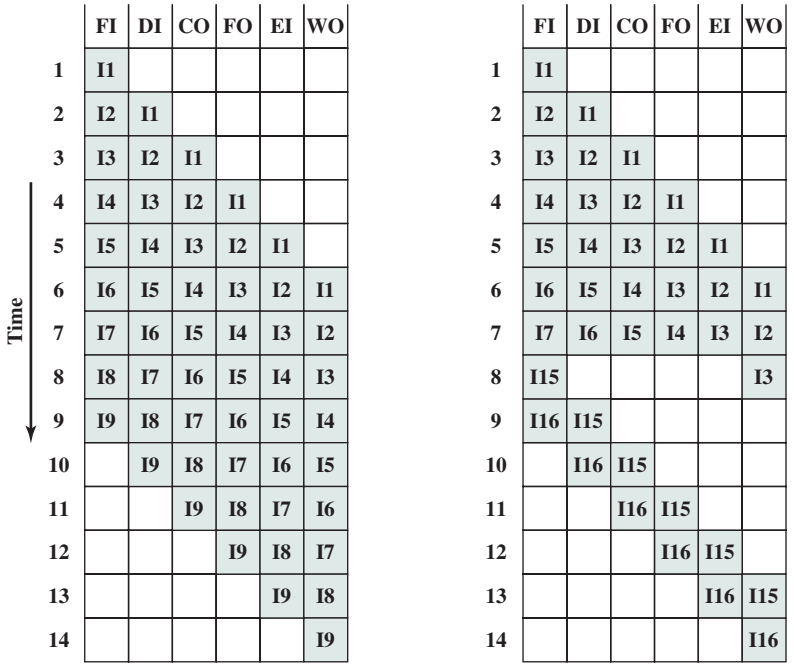
Figure 14.12 Six-Stage CPU Instruction Pipeline

Instruction pipelining is a powerful technique for enhancing performance but requires careful design to achieve optimum results with reasonable complexity.

Pipeline Performance

In this subsection, we develop some simple measures of pipeline performance and relative speedup (based on a discussion in [HWAN93]). The cycle time τ of an **instruction pipeline** is the time needed to advance a set of instructions one stage through the pipeline; each column in Figures 14.10 and 14.11 represents one cycle time. The cycle time can be determined as

$$\tau = \max_i[\tau_i] + d = \tau_m + d \quad 1 \leq i \leq k$$



(a) No branches (b) With conditional branch

Figure 14.13 An Alternative Pipeline Depiction

where

τ_i = time delay of the circuitry in the ith stage of the pipeline
τ_m = maximum stage delay (delay through stage which experiences the largest delay)
k = number of stages in the instruction pipeline
d = time delay of a latch, needed to advance signals and data from one stage to the next

In general, the time delay d is equivalent to a clock pulse and $\tau_m \gg d$. Now suppose that n instructions are processed, with no branches. Let $T_{k,n}$ be the total time required for a pipeline with k stages to execute n instructions. Then

$$T_{k,n} = [k + (n - 1)]\tau \qquad \textbf{(14.1)}$$

A total of k cycles are required to complete the execution of the first instruction, and the remaining $n - 1$ instructions require $n - 1$ cycles.[2] This equation is easily verified from Figure 14.10. The ninth instruction completes at time cycle 14:

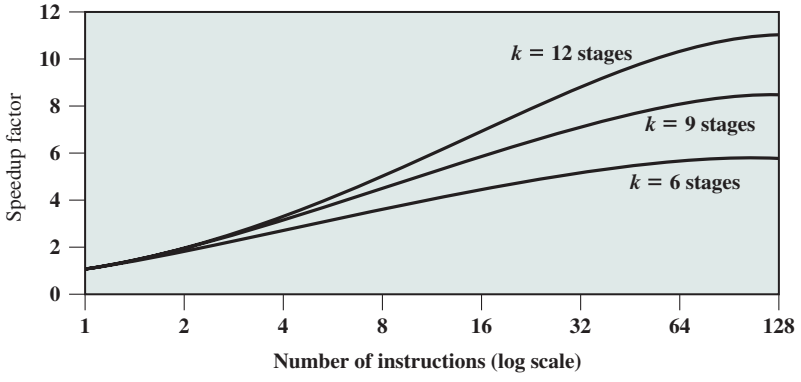
$$14 = [6 + (9 - 1)]$$

[2]We are being a bit sloppy here. The cycle time will only equal the maximum value of τ when all the stages are full. At the beginning, the cycle time may be less for the first one or few cycles.

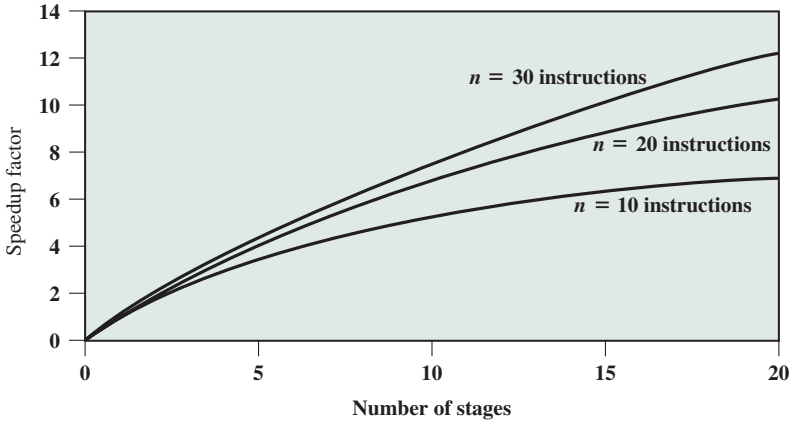
Now consider a processor with equivalent functions but no pipeline, and assume that the instruction cycle time is $k\tau$. The speedup factor for the instruction pipeline compared to execution without the pipeline is defined as

$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{[k + (n - 1)]\tau} = \frac{nk}{k + (n - 1)} \tag{14.2}$$

Figure 14.14a plots the speedup factor as a function of the number of instructions that are executed without a branch. As might be expected, at the limit ($n \rightarrow \infty$), we have a k-fold speedup. Figure 14.14b shows the speedup factor as a function of the number of stages in the instruction pipeline.[3] In this case, the speedup factor approaches the number of instructions that can be fed into the pipeline without branches. Thus, the larger the number of pipeline stages, the greater the potential for speedup. However, as a practical matter, the potential gains of additional



(a)



(b)

Figure 14.14 Speedup Factors with Instruction Pipelining

[3]Note that the x-axis is logarithmic in Figure 14.14a and linear in Figure 14.14b.

pipeline stages are countered by increases in cost, delays between stages, and the fact that branches will be encountered requiring the flushing of the pipeline.

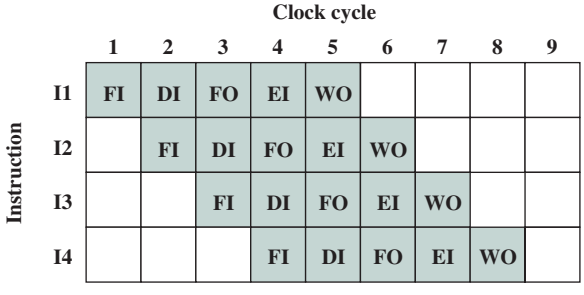
Pipeline Hazards

In the previous subsection, we mentioned some of the situations that can result in less than optimal pipeline performance. In this subsection, we examine this issue in a more systematic way. Chapter 16 revisits this issue, in more detail, after we have introduced the complexities found in superscalar pipeline organizations.

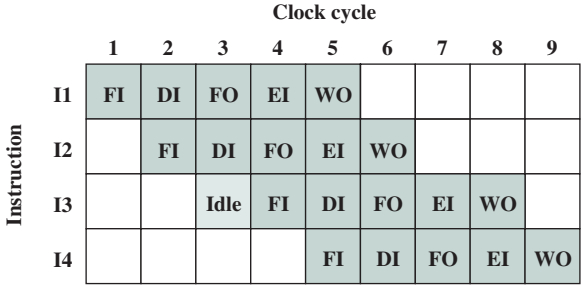
A **pipeline hazard** occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution. Such a pipeline stall is also referred to as a *pipeline bubble*. There are three types of hazards: resource, data, and control.

RESOURCE HAZARDS A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource. The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline. A resource hazard is sometime referred to as a *structural hazard*.

Let us consider a simple example of a resource hazard. Assume a simplified five-stage pipeline, in which each stage takes one clock cycle. Figure 14.15a shows the ideal case, in which a new instruction enters the pipeline each clock cycle. Now assume that main memory has a single port and that all instruction fetches and data reads and writes must be performed one at a time. Further, ignore the cache. In this case, an operand read to or write from memory cannot be performed in parallel



(a) Five-stage pipeline, ideal case



(b) I1 source operand in memory

Figure 14.15 Example of Resource Hazard

with an instruction fetch. This is illustrated in Figure 14.15b, which assumes that the source operand for instruction I1 is in memory, rather than a register. Therefore, the fetch instruction stage of the pipeline must idle for one cycle before beginning the instruction fetch for instruction I3. The figure assumes that all other operands are in registers.

Another example of a resource conflict is a situation in which multiple instructions are ready to enter the execute instruction phase and there is a single ALU. One solutions to such resource hazards is to increase available resources, such as having multiple ports into main memory and multiple ALU units.

Reservation Table Analyzer

One approach to analyzing resource conflicts and aiding in the design of pipelines is the reservation table. We examine reservation tables in Appendix N.

DATA HAZARDS A data hazard occurs when there is a conflict in the access of an operand location. In general terms, we can state the hazard in this form: Two instructions in a program are to be executed in sequence and both access a particular memory or register operand. If the two instructions are executed in strict sequence, no problem occurs. However, if the instructions are executed in a pipeline, then it is possible for the operand value to be updated in such a way as to produce a different result than would occur with strict sequential execution. In other words, the program produces an incorrect result because of the use of pipelining.

As an example, consider the following x86 machine instruction sequence:

```
ADD EAX, EBX /* EAX = EAX + EBX
SUB ECX, EAX /* ECX = ECX - EAX
```

The first instruction adds the contents of the 32-bit registers EAX and EBX and stores the result in EAX. The second instruction subtracts the contents of EAX from ECX and stores the result in ECX. Figure 14.16 shows the pipeline behavior.

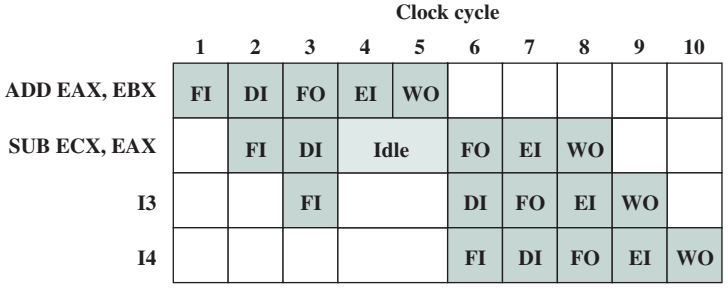


Figure 14.16 Example of Data Hazard

The ADD instruction does not update register EAX until the end of stage 5, which occurs at clock cycle 5. But the SUB instruction needs that value at the beginning of its stage 2, which occurs at clock cycle 4. To maintain correct operation, the pipeline must stall for two clocks cycles. Thus, in the absence of special hardware and specific avoidance algorithms, such a data hazard results in inefficient pipeline usage.

There are three types of data hazards:

- **Read after write (RAW), or true dependency:** An instruction modifies a register or memory location and a succeeding instruction reads the data in that memory or register location. A hazard occurs if the read takes place before the write operation is complete.
- **Write after read (WAR), or antidependency:** An instruction reads a register or memory location and a succeeding instruction writes to the location. A hazard occurs if the write operation completes before the read operation takes place.
- **Write after write (WAW), or output dependency:** Two instructions both write to the same location. A hazard occurs if the write operations take place in the reverse order of the intended sequence.

The example of Figure 14.16 is a RAW hazard. The other two hazards are best discussed in the context of superscalar organization, discussed in Chapter 16.

CONTROL HAZARDS A control hazard, also known as a *branch hazard,* occurs when the pipeline makes the wrong decision on a branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded. We discuss approaches to dealing with control hazards next.

Dealing with Branches

One of the major problems in designing an instruction pipeline is assuring a steady flow of instructions to the initial stages of the pipeline. The primary impediment, as we have seen, is the conditional branch instruction. Until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not.

A variety of approaches have been taken for dealing with conditional branches:

- Multiple streams
- Prefetch branch target
- Loop buffer
- Branch prediction
- Delayed branch

MULTIPLE STREAMS A simple pipeline suffers a penalty for a branch instruction because it must choose one of two instructions to fetch next and may make the wrong choice. A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams. There are two problems with this approach:

- With multiple pipelines there are contention delays for access to the registers and to memory.

- Additional branch instructions may enter the pipeline (either stream) before the original branch decision is resolved. Each such instruction needs an additional stream.

Despite these drawbacks, this strategy can improve performance. Examples of machines with two or more pipeline streams are the IBM 370/168 and the IBM 3033.

PREFETCH BRANCH TARGET When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This target is then saved until the branch instruction is executed. If the branch is taken, the target has already been prefetched.

The IBM 360/91 uses this approach.

LOOP BUFFER A loop buffer is a small, very-high-speed memory maintained by the instruction fetch stage of the pipeline and containing the n most recently fetched instructions, in sequence. If a branch is to be taken, the hardware first checks whether the branch target is within the buffer. If so, the next instruction is fetched from the buffer. The loop buffer has three benefits:

1. With the use of prefetching, the loop buffer will contain some instruction sequentially ahead of the current instruction fetch address. Thus, instructions fetched in sequence will be available without the usual memory access time.
2. If a branch occurs to a target just a few locations ahead of the address of the branch instruction, the target will already be in the buffer. This is useful for the rather common occurrence of IF–THEN and IF–THEN–ELSE sequences.
3. This strategy is particularly well suited to dealing with loops, or iterations; hence the name *loop buffer*. If the loop buffer is large enough to contain all the instructions in a loop, then those instructions need to be fetched from memory only once, for the first iteration. For subsequent iterations, all the needed instructions are already in the buffer.

The loop buffer is similar in principle to a cache dedicated to instructions. The differences are that the loop buffer only retains instructions in sequence and is much smaller in size and hence lower in cost.

Figure 14.17 gives an example of a loop buffer. If the buffer contains 256 bytes, and byte addressing is used, then the least significant 8 bits are used to index the

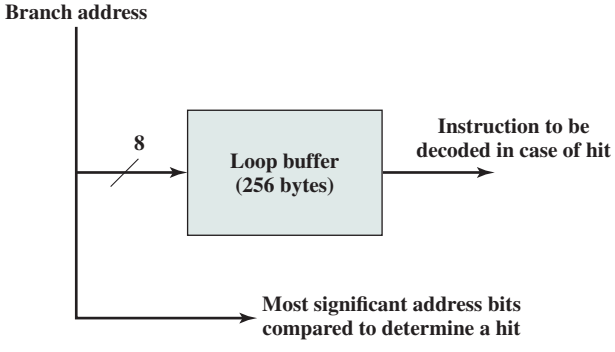


Figure 14.17 Loop Buffer

buffer. The remaining most significant bits are checked to determine if the branch target lies within the environment captured by the buffer.

Among the machines using a loop buffer are some of the CDC machines (Star-100, 6600, 7600) and the CRAY-1. A specialized form of loop buffer is available on the Motorola 68010, for executing a three-instruction loop involving the DBcc (decrement and branch on condition) instruction (see Problem 14.14). A three-word buffer is maintained, and the processor executes these instructions repeatedly until the loop condition is satisfied.

Branch Prediction Simulator
Branch Target Buffer

BRANCH PREDICTION Various techniques can be used to predict whether a branch will be taken. Among the more common are the following:

- Predict never taken
- Predict always taken
- Predict by opcode
- Taken/not taken switch
- Branch history table

The first three approaches are static: they do not depend on the execution history up to the time of the conditional branch instruction. The latter two approaches are dynamic: They depend on the execution history.

The first two approaches are the simplest. These either always assume that the branch will not be taken and continue to fetch instructions in sequence, or they always assume that the branch will be taken and always fetch from the branch target. The predict-never-taken approach is the most popular of all the branch prediction methods.

Studies analyzing program behavior have shown that conditional branches are taken more than 50% of the time [LILJ88], and so if the cost of prefetching from either path is the same, then always prefetching from the branch target address should give better performance than always prefetching from the sequential path. However, in a paged machine, prefetching the branch target is more likely to cause a page fault than prefetching the next instruction in sequence, and so this performance penalty should be taken into account. An avoidance mechanism may be employed to reduce this penalty.

The final static approach makes the decision based on the opcode of the branch instruction. The processor assumes that the branch will be taken for certain branch opcodes and not for others. [LILJ88] reports success rates of greater than 75% with this strategy.

Dynamic branch strategies attempt to improve the accuracy of prediction by recording the history of conditional branch instructions in a program. For example, one or more bits can be associated with each conditional branch instruction that

reflect the recent history of the instruction. These bits are referred to as a taken/not taken switch that directs the processor to make a particular decision the next time the instruction is encountered. Typically, these history bits are not associated with the instruction in main memory. Rather, they are kept in temporary high-speed storage. One possibility is to associate these bits with any conditional branch instruction that is in a cache. When the instruction is replaced in the cache, its history is lost. Another possibility is to maintain a small table for recently executed branch instructions with one or more history bits in each entry. The processor could access the table associatively, like a cache, or by using the low-order bits of the branch instruction's address.

With a single bit, all that can be recorded is whether the last execution of this instruction resulted in a branch or not. A shortcoming of using a single bit appears in the case of a conditional branch instruction that is almost always taken, such as a loop instruction. With only one bit of history, an error in prediction will occur twice for each use of the loop: once on entering the loop, and once on exiting.

If two bits are used, they can be used to record the result of the last two instances of the execution of the associated instruction, or to record a state in some other fashion. Figure 14.18 shows a typical approach (see Problem 14.13 for other

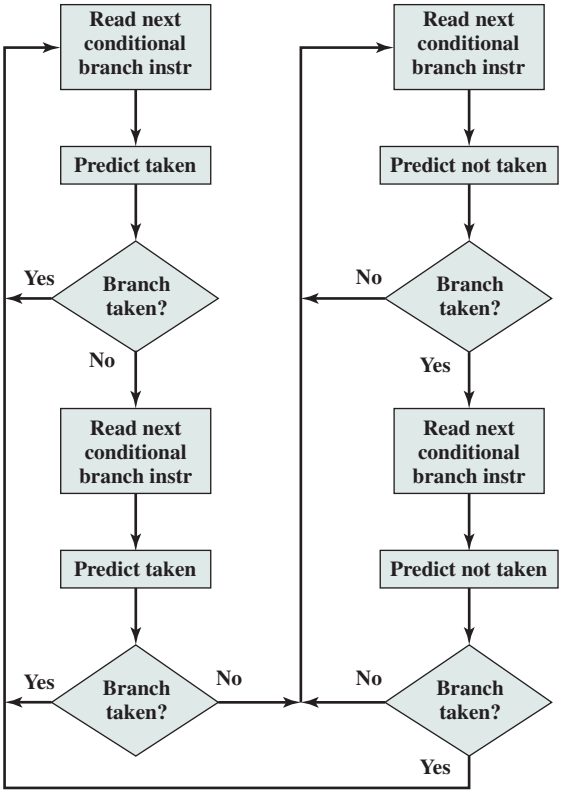


Figure 14.18 Branch Prediction Flowchart

possibilities). Assume that the algorithm starts at the upper-left-hand corner of the flowchart. As long as each succeeding conditional branch instruction that is encountered is taken, the decision process predicts that the next branch will be taken. If a single prediction is wrong, the algorithm continues to predict that the next branch is taken. Only if two successive branches are not taken does the algorithm shift to the right-hand side of the flowchart. Subsequently, the algorithm will predict that branches are not taken until two branches in a row are taken. Thus, the algorithm requires two consecutive wrong predictions to change the prediction decision.

The decision process can be represented more compactly by a finite-state machine, shown in Figure 14.19. The finite-state machine representation is commonly used in the literature.

The use of history bits, as just described, has one drawback: If the decision is made to take the branch, the target instruction cannot be fetched until the target address, which is an operand in the conditional branch instruction, is decoded. Greater efficiency could be achieved if the instruction fetch could be initiated as soon as the branch decision is made. For this purpose, more information must be saved, in what is known as a branch target buffer, or a branch history table.

The branch history table is a small cache memory associated with the instruction fetch stage of the pipeline. Each entry in the table consists of three elements: the address of a branch instruction, some number of history bits that record the state of use of that instruction, and information about the target instruction. In most proposals and implementations, this third field contains the address of the target instruction. Another possibility is for the third field to actually contain the target instruction. The trade-off is clear: Storing the target address yields a smaller table but a greater instruction fetch time compared with storing the target instruction [RECH98].

Figure 14.20 contrasts this scheme with a predict-never-taken strategy. With the former strategy, the instruction fetch stage always fetches the next sequential

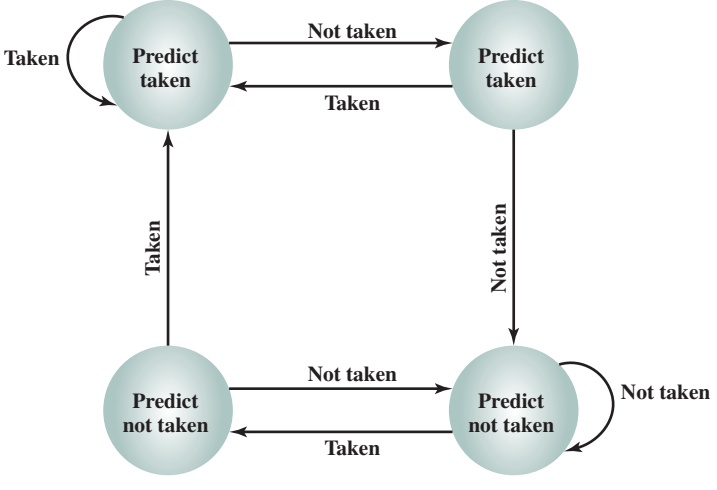


Figure 14.19 Branch Prediction State Diagram

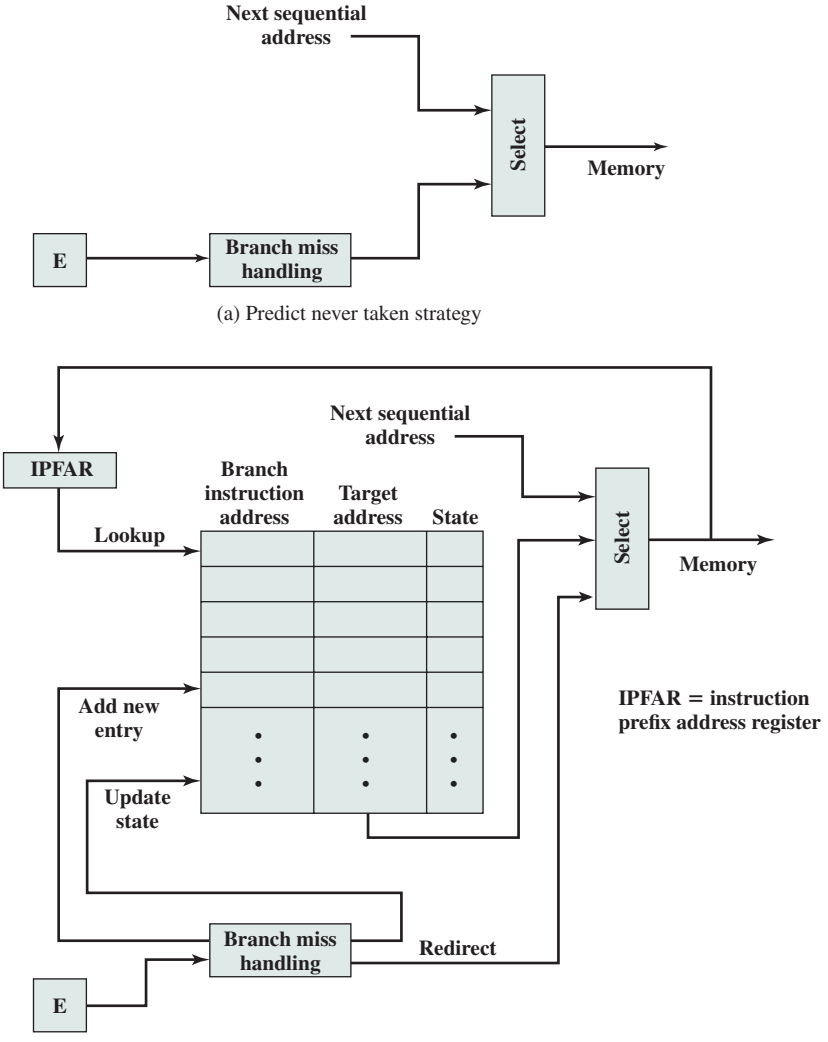


Figure 14.20 Dealing with Branches

address. If a branch is taken, some logic in the processor detects this and instructs that the next instruction be fetched from the target address (in addition to flushing the pipeline). The branch history table is treated as a cache. Each prefetch triggers a lookup in the branch history table. If no match is found, the next sequential address is used for the fetch. If a match is found, a prediction is made based on the state of the instruction: Either the next sequential address or the branch target address is fed to the select logic.

When the branch instruction is executed, the execute stage signals the branch history table logic with the result. The state of the instruction is updated to reflect a correct or incorrect prediction. If the prediction is incorrect, the select logic is

redirected to the correct address for the next fetch. When a conditional branch instruction is encountered that is not in the table, it is added to the table and one of the existing entries is discarded, using one of the cache replacement algorithms discussed in Chapter 4.

A refinement of the branch history approach is referred to as two-level or correlation-based branch history [YEH91]. This approach is based on the assumption that whereas in loop-closing branches, the past history of a particular branch instruction is a good predictor of future behavior, with more complex control-flow structures, the direction of a branch is frequently correlated with the direction of related branches. An example is an if-then-else or case structure. There are a number of strategies possible. Typically, recent global branch history (i.e., the history of the most recent branches not just of this branch instruction) is used in addition to the history of the current branch instruction. The general structure is defined as an (m, n) correlator, which uses the behavior of the last m branches to choose from 2^m n-bit branch predictors for the current branch instruction. In other words, an n-bit history is kept for a give branch for each possible combination of branches taken by the most recent m branches.

DELAYED BRANCH It is possible to improve pipeline performance by automatically rearranging instructions within a program, so that branch instructions occur later than actually desired. This intriguing approach is examined in Chapter 15.

Intel 80486 Pipelining

An instructive example of an instruction pipeline is that of the Intel 80486. The 80486 implements a five-stage pipeline:

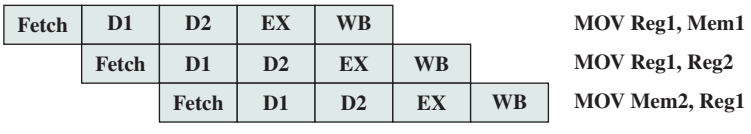
- **Fetch:** Instructions are fetched from the cache or from external memory and placed into one of the two 16-byte prefetch buffers. The objective of the fetch stage is to fill the prefetch buffers with new data as soon as the old data have been consumed by the instruction decoder. Because instructions are of variable length (from 1 to 11 bytes not counting prefixes), the status of the prefetcher relative to the other pipeline stages varies from instruction to instruction. On average, about five instructions are fetched with each 16-byte load [CRAW90]. The fetch stage operates independently of the other stages to keep the prefetch buffers full.
- **Decode stage 1:** All opcode and addressing-mode information is decoded in the D1 stage. The required information, as well as instruction-length information, is included in at most the first 3 bytes of the instruction. Hence, 3 bytes are passed to the D1 stage from the prefetch buffers. The D1 decoder can then direct the D2 stage to capture the rest of the instruction (displacement and immediate data), which is not involved in the D1 decoding.
- **Decode stage 2:** The D2 stage expands each opcode into control signals for the ALU. It also controls the computation of the more complex addressing modes.
- **Execute:** This stage includes ALU operations, cache access, and register update.

- **Write back:** This stage, if needed, updates registers and status flags modified during the preceding execute stage. If the current instruction updates memory, the computed value is sent to the cache and to the bus-interface write buffers at the same time.

With the use of two decode stages, the pipeline can sustain a throughput of close to one instruction per clock cycle. Complex instructions and conditional branches can slow down this rate.

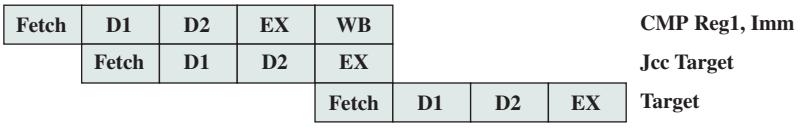
Figure 14.21 shows examples of the operation of the pipeline. Figure 14.21a shows that there is no delay introduced into the pipeline when a memory access is required. However, as Figure 14.21b shows, there can be a delay for values used to compute memory addresses. That is, if a value is loaded from memory into a register and that register is then used as a base register in the next instruction, the processor will stall for one cycle. In this example, the processor accesses the cache in the EX stage of the first instruction and stores the value retrieved in the register during the WB stage. However, the next instruction needs this register in its D2 stage. When the D2 stage lines up with the WB stage of the previous instruction, bypass signal paths allow the D2 stage to have access to the same data being used by the WB stage for writing, saving one pipeline stage.

Figure 14.21c illustrates the timing of a branch instruction, assuming that the branch is taken. The compare instruction updates condition codes in the WB stage, and bypass paths make this available to the EX stage of the jump instruction at the same time. In parallel, the processor runs a speculative fetch cycle to the target of the jump during the EX stage of the jump instruction. If the processor determines a false branch condition, it discards this prefetch and continues execution with the next sequential instruction (already fetched and decoded).



(a) No data load delay in the pipeline

(b) Pointer load delay



(c) Branch instruction timing

Figure 14.21 80486 Instruction Pipeline Examples

14.5 THE x86 PROCESSOR FAMILY

The x86 organization has evolved dramatically over the years. In this section we examine some of the details of the most recent processor organizations, concentrating on common elements in single processors. Chapter 16 looks at superscalar aspects of the x86, and Chapter 18 examines the multicore organization. An overview of the Pentium 4 processor organization is depicted in Figure 4.18.

Register Organization

The register organization includes the following types of registers (Table 14.2):

- **General:** There are eight 32-bit general-purpose registers (see Figure 14.3c). These may be used for all types of x86 instructions; they can also hold operands for address calculations. In addition, some of these registers also serve special purposes. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands without having to reference these registers explicitly in the instruction. As a result, a number of instructions can be

Table 14.2 x86 Processor Registers

(a) Integer Unit in 32-bit Mode

Type	Number	Length (bits)	Purpose
General	8	32	General-purpose user registers
Segment	6	16	Contain segment selectors
EFLAGS	1	32	Status and control bits
Instruction Pointer	1	32	Instruction pointer

(b) Integer Unit in 64-bit Mode

Type	Number	Length (bits)	Purpose
General	16	32	General-purpose user registers
Segment	6	16	Contain segment selectors
RFLAGS	1	64	Status and control bits
Instruction Pointer	1	64	Instruction pointer

(c) Floating-Point Unit

Type	Number	Length (bits)	Purpose
Numeric	8	80	Hold floating-point numbers
Control	1	16	Control bits
Status	1	16	Status bits
Tag Word	1	16	Specifies contents of numeric registers
Instruction Pointer	1	48	Points to instruction interrupted by exception
Data Pointer	1	48	Points to operand interrupted by exception

encoded more compactly. In 64-bit mode, there are sixteen 64-bit general-purpose registers.

- **Segment:** The six 16-bit segment registers contain segment selectors, which index into segment tables, as discussed in Chapter 8. The code segment (CS) register references the segment containing the instruction being executed. The stack segment (SS) register references the segment containing a user-visible stack. The remaining segment registers (DS, ES, FS, GS) enable the user to reference up to four separate data segments at a time.
- **Flags:** The 32-bit EFLAGS register contains condition codes and various mode bits. In 64-bit mode, this register is extended to 64 bits and referred to as RFLAGS. In the current architecture definition, the upper 32 bits of RFLAGS are unused.
- **Instruction pointer:** Contains the address of the current instruction.

There are also registers specifically devoted to the floating-point unit:

- **Numeric:** Each register holds an extended-precision 80-bit floating-point number. There are eight registers that function as a stack, with push and pop operations available in the instruction set.
- **Control:** The 16-bit control register contains bits that control the operation of the floating-point unit, including the type of rounding control; single, double, or extended precision; and bits to enable or disable various exception conditions.
- **Status:** The 16-bit status register contains bits that reflect the current state of the floating-point unit, including a 3-bit pointer to the top of the stack; condition codes reporting the outcome of the last operation; and exception flags.
- **Tag word:** This 16-bit register contains a 2-bit tag for each floating-point numeric register, which indicates the nature of the contents of the corresponding register. The four possible values are valid, zero, special (NaN, infinity, denormalized), and empty. These tags enable programs to check the contents of a numeric register without performing complex decoding of the actual data in the register. For example, when a context switch is made, the processor need not save any floating-point registers that are empty.

The use of most of the aforementioned registers is easily understood. Let us elaborate briefly on several of the registers.

EFLAGS REGISTER The EFLAGS register (Figure 14.22) indicates the condition of the processor and helps to control its operation. It includes the six condition codes defined in Table 12.9 (carry, parity, auxiliary, zero, sign, overflow), which report the results of an integer operation. In addition, there are bits in the register that may be referred to as control bits:

- **Trap flag (TF):** When set, causes an interrupt after the execution of each instruction. This is used for debugging.
- **Interrupt enable flag (IF):** When set, the processor will recognize external interrupts.

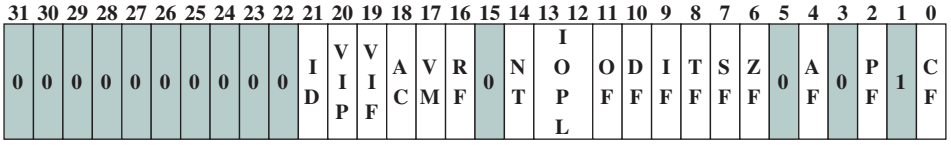
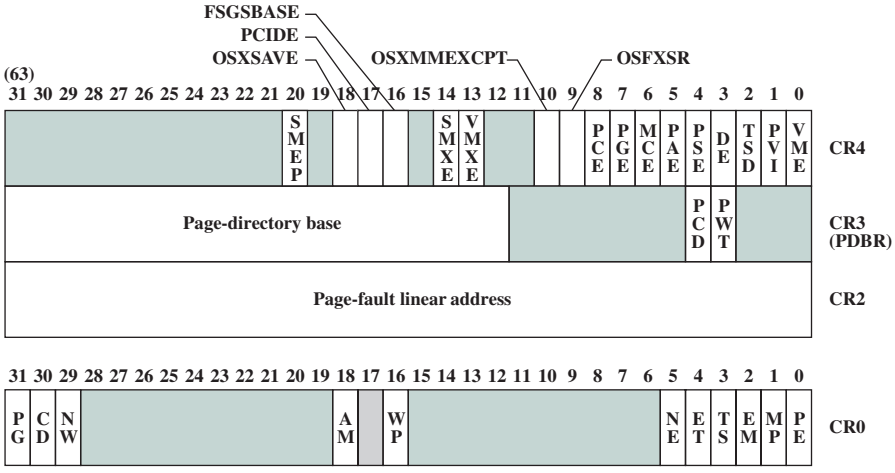

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID = Identification flag
X VIP = Virtual interrupt pending
X VIF = Virtual interrupt flag
X AC = Alignment check
X VM = Virtual 8086 mode
X RF = Resume flag
X NT = Nested task flag
X IOPL = I/O privilege level
S OF = Overflow flag
C DF = Direction flag
X IF = Interrupt enable flag
X TF = Trap flag
S SF = Sign flag
S ZF = Zero flag
S AF = Auxiliary carry flag
S PF = Parity flag
S CF = Carry flag
S indicates a status flag.
C indicates a control flag.
X indicates a system flag.
Shaded bits are reserved.

Figure 14.22 x86 EFLAGS Register

- **Direction flag (DF):** Determines whether string processing instructions increment or decrement the 16-bit half-registers SI and DI (for 16-bit operations) or the 32-bit registers ESI and EDI (for 32-bit operations).
- **I/O privilege flag (IOPL):** When set, causes the processor to generate an exception on all accesses to I/O devices during protected-mode operation.
- **Resume flag (RF):** Allows the programmer to disable debug exceptions so that the instruction can be restarted after a debug exception without immediately causing another debug exception.
- **Alignment check (AC):** Activates if a word or doubleword is addressed on a nonword or nondoubleword boundary.
- **Identification flag (ID):** If this bit can be set and cleared, then this processor supports the processorID instruction. This instruction provides information about the vendor, family, and model.

In addition, there are 4 bits that relate to operating mode. The Nested Task (NT) flag indicates that the current task is nested within another task in protected-mode operation. The Virtual Mode (VM) bit allows the programmer to enable or disable virtual 8086 mode, which determines whether the processor runs as an 8086 machine. The Virtual Interrupt Flag (VIF) and Virtual Interrupt Pending (VIP) flag are used in a multitasking environment.

CONTROL REGISTERS The x86 employs four control registers (register CR1 is unused) to control various aspects of processor operation (Figure 14.23). All of the registers except CR0 are either 32 bits or 64 bits long, depending on whether the implementation supports the x86 64-bit architecture. The CR0 register contains system control flags, which control modes or indicate states that apply generally



Shaded area indicates reserved bits.

OSXSAVE	=	XSAVE enable bit	VME	=	Virtual 8086 mode extensions
PCIDE	=	Enables process-context identifiers	PCD	=	Page-level cache disable
FSGSBASE	=	Enables segment base instructions	PWT	=	Page-level writes transparent
SMXE	=	Enable safer mode extensions	PG	=	Paging
VMXE	=	Enable virtual machine extensions	CD	=	Cache disable
OSXMMEXCPT	=	Support unmasked SIMD FP exceptions	NW	=	Not write through
OSFXSR	=	Support FXSAVE, FXSTOR	AM	=	Alignment mask
PCE	=	Performance counter enable	WP	=	Write protect
PGE	=	Page global enable	NE	=	Numeric error
MCE	=	Machine check enable	ET	=	Extension type
PAE	=	Physical address extension	TS	=	Task switched
PSE	=	Page size extensions	EM	=	Emulation
DE	=	Debug extensions	MP	=	Monitor coprocessor
TSD	=	Time stamp disable	PE	=	Protection enable
PVI	=	Protected mode virtual interrupt			

Figure 14.23 x86 Control Registers

to the processor rather than to the execution of an individual task. The flags are as follows:

- **Protection Enable (PE):** Enable/disable protected mode of operation.
- **Monitor Coprocessor (MP):** Only of interest when running programs from earlier machines on the x86; it relates to the presence of an arithmetic coprocessor.
- **Emulation (EM):** Set when the processor does not have a floating-point unit, and causes an interrupt when an attempt is made to execute floating-point instructions.
- **Task Switched (TS):** Indicates that the processor has switched tasks.
- **Extension Type (ET):** Not used on the Pentium and later machines; used to indicate support of math coprocessor instructions on earlier machines.

- **Numeric Error (NE):** Enables the standard mechanism for reporting floating-point errors on external bus lines.
- **Write Protect (WP):** When this bit is clear, read-only user-level pages can be written by a supervisor process. This feature is useful for supporting process creation in some operating systems.
- **Alignment Mask (AM):** Enables/disables alignment checking.
- **Not Write Through (NW):** Selects mode of operation of the data cache. When this bit is set, the data cache is inhibited from cache write-through operations.
- **Cache Disable (CD):** Enables/disables the internal cache fill mechanism.
- **Paging (PG):** Enables/disables paging.

When paging is enabled, the CR2 and CR3 registers are valid. The CR2 register holds the 32-bit linear address of the last page accessed before a page fault interrupt. The leftmost 20 bits of CR3 hold the 20 most significant bits of the base address of the page directory; the remainder of the address contains zeros. Two bits of CR3 are used to drive pins that control the operation of an external cache. The page-level cache disable (PCD) enables or disables the external cache, and the page-level writes transparent (PWT) bit controls write through in the external cache. CR4 contains additional control bits.

MMX REGISTERS Recall from Section 10.3 that the x86 MMX capability makes use of several 64-bit data types. The MMX instructions make use of 3-bit register address fields, so that eight MMX registers are supported. In fact, the processor does not include specific MMX registers. Rather, the processor uses an aliasing technique (Figure 14.24). The existing floating-point registers are used to store MMX values. Specifically, the low-order 64 bits (mantissa) of each floating-point register are used to form the eight MMX registers. Thus, the older 32-bit x86 architecture is easily extended to support the MMX capability. Some key characteristics of the MMX use of these registers are as follows:

- Recall that the floating-point registers are treated as a stack for floating-point operations. For MMX operations, these same registers are accessed directly.
- The first time that an MMX instruction is executed after any floating-point operations, the FP tag word is marked valid. This reflects the change from stack operation to direct register addressing.
- The EMMS (Empty MMX State) instruction sets bits of the FP tag word to indicate that all registers are empty. It is important that the programmer insert this instruction at the end of an MMX code block so that subsequent floating-point operations function properly.
- When a value is written to an MMX register, bits [79:64] of the corresponding FP register (sign and exponent bits) are set to all ones. This sets the value in the FP register to NaN (not a number) or infinity when viewed as a floating-point value. This ensures that an MMX data value will not look like a valid floating-point value.

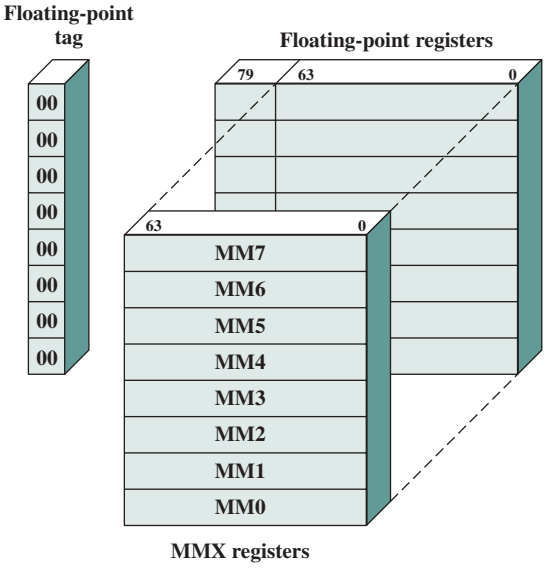


Figure 14.24 Mapping of MMX Registers to Floating-Point Registers

Interrupt Processing

Interrupt processing within a processor is a facility provided to support the operating system. It allows an application program to be suspended, in order that a variety of interrupt conditions can be serviced and later resumed.

INTERRUPTS AND EXCEPTIONS Two classes of events cause the x86 to suspend execution of the current instruction stream and respond to the event: interrupts and exceptions. In both cases, the processor saves the context of the current process and transfers to a predefined routine to service the condition. An *interrupt* is generated by a signal from hardware, and it may occur at random times during the execution of a program. An *exception* is generated from software, and it is provoked by the execution of an instruction. There are two sources of interrupts and two sources of exceptions:

1. **Interrupts**
   - **Maskable interrupts:** Received on the processor's INTR pin. The processor does not recognize a maskable interrupt unless the interrupt enable flag (IF) is set.
   - **Nonmaskable interrupts:** Received on the processor's NMI pin. Recognition of such interrupts cannot be prevented.
2. **Exceptions**
   - **Processor-detected exceptions:** Results when the processor encounters an error while attempting to execute an instruction.

- **Programmed exceptions:** These are instructions that generate an exception (e.g., INTO, INT3, INT, and BOUND).

INTERRUPT VECTOR TABLE Interrupt processing on the x86 uses the interrupt vector table. Every type of interrupt is assigned a number, and this number is used to index into the interrupt vector table. This table contains 256 32-bit interrupt vectors, which is the address (segment and offset) of the interrupt service routine for that interrupt number.

Table 14.3 shows the assignment of numbers in the interrupt vector table; shaded entries represent interrupts, while nonshaded entries are exceptions. The NMI hardware interrupt is type 2. INTR hardware interrupts are assigned numbers in the range of 32 to 255; when an INTR interrupt is generated, it must be accompanied on the bus with the interrupt vector number for this interrupt. The remaining vector numbers are used for exceptions.

If more than one exception or interrupt is pending, the processor services them in a predictable order. The location of vector numbers within the table does not reflect priority. Instead, priority among exceptions and interrupts is organized into five classes. In descending order of priority, these are

- **Class 1:** Traps on the previous instruction (vector number 1)
- **Class 2:** External interrupts (2, 32–255)
- **Class 3:** Faults from fetching next instruction (3, 14)
- **Class 4:** Faults from decoding the next instruction (6, 7)
- **Class 5:** Faults on executing an instruction (0, 4, 5, 8, 10–14, 16, 17)

INTERRUPT HANDLING Just as with a transfer of execution using a CALL instruction, a transfer to an interrupt-handling routine uses the system stack to store the processor state. When an interrupt occurs and is recognized by the processor, a sequence of events takes place:

1. If the transfer involves a change of privilege level, then the current stack segment register and the current extended stack pointer (ESP) register are pushed onto the stack.
2. The current value of the EFLAGS register is pushed onto the stack.
3. Both the interrupt (IF) and trap (TF) flags are cleared. This disables INTR interrupts and the trap or single-step feature.
4. The current code segment (CS) pointer and the current instruction pointer (IP or EIP) are pushed onto the stack.
5. If the interrupt is accompanied by an error code, then the error code is pushed onto the stack.
6. The interrupt vector contents are fetched and loaded into the CS and IP or EIP registers. Execution continues from the interrupt service routine.

To return from an interrupt, the interrupt service routine executes an IRET instruction. This causes all of the values saved on the stack to be restored; execution resumes from the point of the interrupt.

Table 14.3 x86 Exception and Interrupt Vector Table

Vector Number	Description
0	Divide error; division overflow or division by zero
1	Debug exception; includes various faults and traps related to debugging
2	NMI pin interrupt; signal on NMI pin
3	Breakpoint; caused by INT 3 instruction, which is a 1-byte instruction useful for debugging
4	INTO-detected overflow; occurs when the processor executes INTO with the OF flag set
5	BOUND range exceeded; the BOUND instruction compares a register with boundaries stored in memory and generates an interrupt if the contents of the register is out of bounds
6	Undefined opcode
7	Device not available; attempt to use ESC or WAIT instruction fails due to lack of external device
8	Double fault; two interrupts occur during the same instruction and cannot be handled serially
9	Reserved
10	Invalid task state segment; segment describing a requested task is not initialized or not valid
11	Segment not present; required segment not present
12	Stack fault; limit of stack segment exceeded or stack segment not present
13	General protection; protection violation that does not cause another exception (e.g., writing to a read-only segment)
14	Page fault
15	Reserved
16	Floating-point error; generated by a floating-point arithmetic instruction
17	Alignment check; access to a word stored at an odd byte address or a doubleword stored at an address not a multiple of 4
18	Machine check; model specific
19–31	Reserved
32–255	User interrupt vectors; provided when INTR signal is activated

Unshaded: exceptions
Shaded: interrupts

14.6 THE ARM PROCESSOR

In this section, we look at some of the key elements of the ARM architecture and organization. We defer a discussion of more complex aspects of organization and pipelining until Chapter 16. For the discussion in this section and in Chapter 16, it is useful to keep in mind key characteristics of the ARM architecture. ARM is primarily a RISC system with the following notable attributes:

- A moderate array of uniform registers, more than are found on some CISC systems but fewer than are found on many RISC systems.
- A load/store model of data processing, in which operations only perform on operands in registers and not directly in memory. All data must be loaded into registers before an operation can be performed; the result can then be used for further processing or stored into memory.
- A uniform fixed-length instruction of 32 bits for the standard set and 16 bits for the Thumb instruction set.
- To make each data processing instruction more flexible, either a shift or rotation can preprocess one of the source registers. To efficiently support this feature, there are separate arithmetic logic unit (ALU) and shifter units.
- A small number of addressing modes with all load/store addressees determined from registers and instruction fields. Indirect or indexed addressing involving values in memory are not used.
- Auto-increment and auto-decrement addressing modes are used to improve the operation of program loops.
- Conditional execution of instructions minimizes the need for conditional branch instructions, thereby improving pipeline efficiency, because pipeline flushing is reduced.

Processor Organization

The ARM processor organization varies substantially from one implementation to the next, particularly when based on different versions of the ARM architecture. However, it is useful for the discussion in this section to present a simplified, generic ARM organization, which is illustrated in Figure 14.25. In this figure, the arrows indicate the flow of data. Each box represents a functional hardware unit or a storage unit.

Data are exchanged with the processor from external memory through a data bus. The value transferred is either a data item, as a result of a load or store instruction, or an instruction fetch. Fetched instructions pass through an instruction decoder before execution, under control of a control unit. The latter includes pipeline logic and provides control signals (not shown) to all the hardware elements of the processor. Data items are placed in the register file, consisting of a set of 32-bit registers. Byte or halfword items treated as twos-complement numbers are sign-extended to 32 bits.

ARM data processing instructions typically have two source registers, *Rn* and *Rm*, and a single result or destination register, *Rd*. The source register values feed into the ALU or a separate multiply unit that makes use of an additional register to accumulate partial results. The ARM processor also includes a hardware unit that can shift or rotate the *Rm* value before it enters the ALU. This shift or rotate occurs within the cycle time of the instruction and increases the power and flexibility of many data processing operations.

The results of an operation are fed back to the destination register. Load/store instructions may also use the output of the arithmetic units to generate the memory address for a load or store.

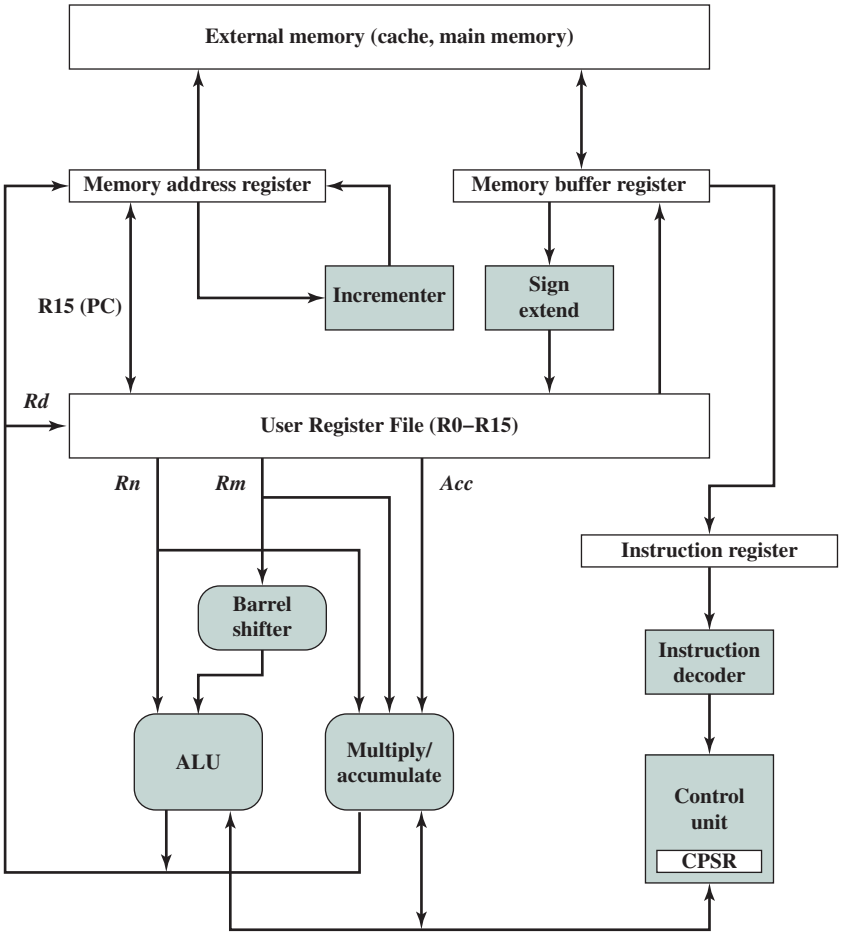


Figure 14.25 Simplified ARM Organization

Processor Modes

It is quite common for a processor to support only a small number of processor modes. For example, many operating systems make use of just two modes: a user mode and a kernel mode, with the latter mode used to execute privileged system software. In contrast, the ARM architecture provides a flexible foundation for operating systems to enforce a variety of protection policies.

The ARM architecture supports seven execution modes. Most application programs execute in **user mode**. While the processor is in user mode, the program being executed is unable to access protected system resources or to change mode, other than by causing an exception to occur.

The remaining six execution modes are referred to as privileged modes. These modes are used to run system software. There are two principal advantages to defining so many different privileged modes: (1) The OS can tailor the use of system software to a variety of circumstances, and (2) certain registers are dedicated for use for each of the privileged modes, allowing swifter changes in context.

The exception modes have full access to system resources and can change modes freely. Five of these modes are known as exception modes. These are entered when specific exceptions occur. Each of these modes has some dedicated registers that substitute for some of the user mode registers, and which are used to avoid corrupting User mode state information when the exception occurs. The exception modes are as follows:

- **Supervisor mode:** Usually what the OS runs in. It is entered when the processor encounters a software interrupt instruction. Software interrupts are a standard way to invoke operating system services on ARM.
- **Abort mode:** Entered in response to memory faults.
- **Undefined mode:** Entered when the processor attempts to execute an instruction that is supported neither by the main integer core nor by one of the coprocessors.
- **Fast interrupt mode:** Entered whenever the processor receives an interrupt signal from the designated fast interrupt source. A fast interrupt cannot be interrupted, but a fast interrupt may interrupt a normal interrupt.
- **Interrupt mode:** Entered whenever the processor receives an interrupt signal from any other interrupt source (other than fast interrupt). An interrupt may only be interrupted by a fast interrupt.

The remaining privileged mode is the **System mode**. This mode is not entered by any exception and uses the same registers available in User mode. The System mode is used for running certain privileged operating system tasks. System mode tasks may be interrupted by any of the five exception categories.

Register Organization

Figure 14.26 depicts the user-visible registers for the ARM. The ARM processor has a total of 37 32-bit registers, classified as follows:

- Thirty-one registers referred to in the ARM manual as general-purpose registers. In fact, some of these, such as the program counters, have special purposes.
- Six program status registers.

Registers are arranged in partially overlapping banks, with the current processor mode determining which bank is available. At any time, sixteen numbered registers and one or two program status registers are visible, for a total of 17 or 18 software-visible registers. Figure 14.26 is interpreted as follows:

- Registers R0 through R7, register R15 (the program counter) and the current program status register (CPSR) are visible in and shared by all modes.
- Registers R8 through R12 are shared by all modes except fast interrupt, which has its own dedicated registers R8_fiq through R12_fiq.
- All the exception modes have their own versions of registers R13 and R14.
- All the exception modes have a dedicated saved program status register (SPSR).

Modes						
	Privileged modes					
		Exception modes				
User	**System**	**Supervisor**	**Abort**	**Undefined**	**Interrupt**	**Fast interrupt**
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13(SP)	R13(SP)	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14(LR)	R14(LR)	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Shading indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode.
SP = stack pointer **CPSR = current program status register**
LR = link register **SPSR = saved program status register**
PC = program counter

Figure 14.26 ARM Register Organization

GENERAL-PURPOSE REGISTERS Register R13 is normally used as a stack pointer and is also known as the SP. Because each exception mode has a separate R13, each exception mode can have its own dedicated program stack. R14 is known as the link register (LR) and is used to hold subroutine return addresses and exception mode returns. Register R15 is the program counter (PC).

PROGRAM STATUS REGISTERS The CPSR is accessible in all processor modes. Each exception mode also has a dedicated SPSR that is used to preserve the value of the CPSR when the associated exception occurs.

The 16 most significant bits of the CPSR contain user flags visible in User mode, and which can be used to affect the operation of a program (Figure 14.27). These are as follows:

- **Condition code flags:** The N, Z, C, and V flags, which are discussed in Chapter 12.
- **Q flag:** used to indicate whether overflow and/or saturation has occurred in some SIMD-oriented instructions.
- **J bit:** indicates the use of special 8-bit instructions, known as Jazelle instructions, which are beyond the scope of our discussion.
- **GE[3:0] bits:** SIMD instructions use bits [19:16] as Greater than or Equal (GE) flags for individual bytes or halfwords of the result.

The 16 least significant bits of the CPSR contain system control flags that can only be altered when the processor is in a privileged mode. The fields are as follows:

- **E bit:** Controls load and store endianness for data; ignored for instruction fetches.
- **Interrupt disable bits:** The A bit disables imprecise data aborts when set; the I bit disables IRQ interrupts when set; and the F bit disables FIQ interrupts when set.
- **T bit:** Indicates whether instructions should be interpreted as normal ARM instructions or Thumb instructions.
- **Mode bits:** Indicates the processor mode.

Interrupt Processing

As with any processor, the ARM includes a facility that enables the processor to interrupt the currently executing program to deal with exception conditions. Exceptions are generated by internal and external sources to cause the processor to handle an event. The processor state just before handling the exception is normally preserved so that the original program can be resumed when the exception routine has completed. More than one exception can arise at the same time. The ARM architecture supports seven types of exceptions. Table 14.4 lists the types of exception and the processor mode that is used to process each type. When an exception occurs, execution is forced from a fixed memory address corresponding to the type of exception. These fixed addresses are called the exception vectors.

If more than one interrupt is outstanding, they are handled in priority order. Table 14.4 lists the exceptions in priority order, highest to lowest.

When an exception occurs, the processor halts execution after the current instruction. The state of the processor is preserved in the SPSR that corresponds to

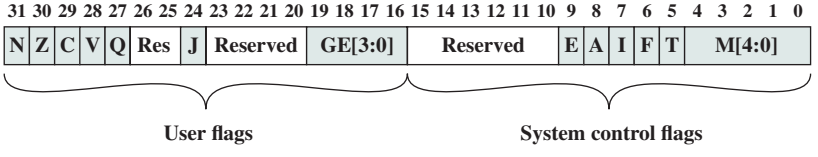

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
N Z C V Q Res J Reserved GE[3:0] Reserved E A I F T M[4:0]
User flags
System control flags

Figure 14.27 Format of ARM CPSR and SPSR

Table 14.4 ARM Interrupt Vector

Exception type	Mode	Normal entry address	Description
Reset	Supervisor	0x00000000	Occurs when the system is initialized.
Data abort	Abort	0x00000010	Occurs when an invalid memory address has been accessed, such as if there is no physical memory for an address or the correct access permission is lacking.
FIQ (fast interrupt)	FIQ	0x0000001C	Occurs when an external device asserts the FIQ pin on the processor. An interrupt cannot be interrupted except by an FIQ. FIQ is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications, therefore minimizing the overhead of context switching. A fast interrupt cannot be interrupted.
IRQ (interrupt)	IRQ	0x00000018	Occurs when an external device asserts the IRQ pin on the processor. An interrupt cannot be interrupted except by an FIQ.
Prefetch abort	Abort	0x0000000C	Occurs when an attempt to fetch an instruction results in a memory fault. The exception is raised when the instruction enters the execute stage of the pipeline.
Undefined instructions	Undefined	0x00000004	Occurs when an instruction not in the instruction set reaches the execute stage of the pipeline.
Software interrupt	Supervisor	0x00000008	Generally used to allow user mode programs to call the OS. The user program executes a SWI instruction with an argument that identifies the function the user wishes to perform.

the type of exception, so that the original program can be resumed when the exception routine has completed. The address of the instruction the processor was just about to execute is placed in the link register of the appropriate processor mode. To return after handling the exception, the SPSR is moved into the CPSR and R14 is moved into the PC.

14.7 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

branch prediction	flag	instruction prefetch
condition code	instruction cycle	program status word (PSW)
delayed branch	instruction pipeline	

Review Questions

14.1 What general roles are performed by processor registers?

14.2 What categories of data are commonly supported by user-visible registers?

14.3 What is the function of condition codes?

14.4 What is a program status word?

14.5 Why is a two-stage instruction pipeline unlikely to cut the instruction cycle time in half, compared with the use of no pipeline?

14.6 List and briefly explain various ways in which an instruction pipeline can deal with conditional branch instructions.

14.7 How are history bits used for branch prediction?

Problems

14.1 **a.** If the last operation performed on a computer with an 8-bit word was an addition in which the two operands were 00000010 and 00000011, what would be the value of the following flags?
- Carry
- Zero
- Overflow
- Sign
- Even Parity
- Half-Carry

b. Repeat for the addition of −1 (twos complement) and +1.

14.2 Repeat Problem 14.1 for the operation A−B, where A contains 11110000 and B contains 0010100.

14.3 A microprocessor is clocked at a rate of 5 GHz.
a. How long is a clock cycle?
b. What is the duration of a particular type of machine instruction consisting of three clock cycles?

14.4 A microprocessor provides an instruction capable of moving a string of bytes from one area of memory to another. The fetching and initial decoding of the instruction takes 10 clock cycles. Thereafter, it takes 15 clock cycles to transfer each byte. The microprocessor is clocked at a rate of 10 GHz.
a. Determine the length of the instruction cycle for the case of a string of 64 bytes.
b. What is the worst-case delay for acknowledging an interrupt if the instruction is noninterruptible?
c. Repeat part (b) assuming the instruction can be interrupted at the beginning of each byte transfer.

14.5 The Intel 8088 consists of a bus interface unit (BIU) and an execution unit (EU), which form a 2-stage pipeline. The BIU fetches instructions into a 4-byte instruction queue. The BIU also participates in address calculations, fetches operands, and writes results in memory as requested by the EU. If no such requests are outstanding and the bus is free, the BIU fills any vacancies in the instruction queue. When the EU completes execution of an instruction, it passes any results to the BIU (destined for memory or I/O) and proceeds to the next instruction.
a. Suppose the tasks performed by the BIU and EU take about equal time. By what factor does pipelining improve the performance of the 8088? Ignore the effect of branch instructions.
b. Repeat the calculation assuming that the EU takes twice as long as the BIU.

14.6 Assume an 8088 is executing a program in which the probability of a program jump is 0.1. For simplicity, assume that all instructions are 2 bytes long.

a. What fraction of instruction fetch bus cycles is wasted?
b. Repeat if the instruction queue is 8 bytes long.

14.7 Consider the timing diagram of Figures 14.10. Assume that there is only a two-stage pipeline (fetch, execute). Redraw the diagram to show how many time units are now needed for four instructions.

14.8 Assume a pipeline with four stages: fetch instruction (FI), decode instruction and calculate addresses (DA), fetch operand (FO), and execute (EX). Draw a diagram similar to Figure 14.10 for a sequence of 7 instructions, in which the third instruction is a branch that is taken and in which there are no data dependencies.

14.9 A pipelined processor has a clock rate of 2.5 GHz and executes a program with 1.5 million instructions. The pipeline has five stages, and instructions are issued at a rate of one per clock cycle. Ignore penalties due to branch instructions and out-of-sequence executions.
a. What is the speedup of this processor for this program compared to a nonpipelined processor, making the same assumptions used in Section 14.4?
b. What is throughput (in MIPS) of the pipelined processor?

14.10 A nonpipelined processor has a clock rate of 2.5 GHz and an average CPI (cycles per instruction) of 4. An upgrade to the processor introduces a five-stage pipeline. However, due to internal pipeline delays, such as latch delay, the clock rate of the new processor has to be reduced to 2 GHz.
a. What is the speedup achieved for a typical program?
b. What is the MIPS rate for each processor?

14.11 Consider an instruction sequence of length n that is streaming through the instruction pipeline. Let p be the probability of encountering a conditional or unconditional branch instruction, and let q be the probability that execution of a branch instruction I causes a jump to a nonconsecutive address. Assume that each such jump requires the pipeline to be cleared, destroying all ongoing instruction processing, when I emerges from the last stage. Revise Equations (14.1) and (14.2) to take these probabilities into account.

14.12 One limitation of the multiple-stream approach to dealing with branches in a pipeline is that additional branches will be encountered before the first branch is resolved. Suggest two additional limitations or drawbacks.

14.13 Consider the state diagrams of Figure 14.28.
a. Describe the behavior of each.
b. Compare these with the branch prediction state diagram in Section 14.4. Discuss the relative merits of each of the three approaches to branch prediction.

14.14 The Motorola 680x0 machines include the instruction Decrement and Branch According to Condition, which has the following form:

DBcc Dn, displacement

where cc is one of the testable conditions, Dn is a general-purpose register, and displacement specifies the target address relative to the current address. The instruction can be defined as follows:

```
if (cc = False)
then begin
     Dn: = (Dn) −1;
     if Dn ≠ −1 then PC: = (PC) + displacement end
else PC: = (PC) + 2;
```

When the instruction is executed, the condition is first tested to determine whether the termination condition for the loop is satisfied. If so, no operation is performed and execution continues at the next instruction in sequence. If the condition is false, the specified data register is decremented and checked to see if it is less than zero. If it is

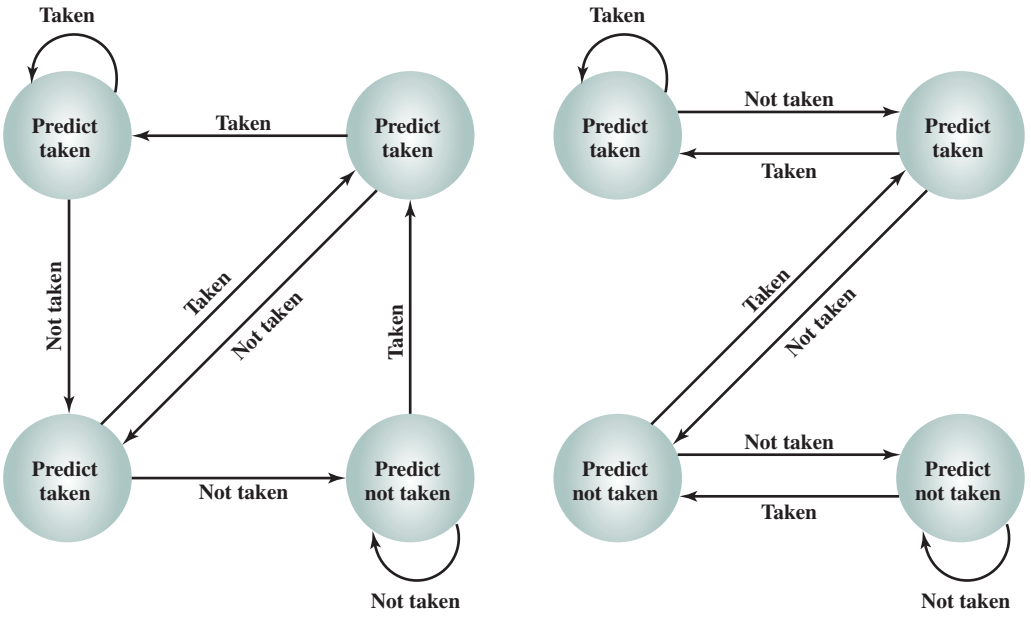


Figure 14.28 Two Branch Prediction State Diagrams

less than zero, the loop is terminated and execution continues at the next instruction in sequence. Otherwise, the program branches to the specified location. Now consider the following assembly-language program fragment:

```
AGAIN   CMPM.L   (A0)+, (A1)+
        DBNE     D1, AGAIN
        NOP
```

Two strings addressed by A0 and A1 are compared for equality; the string pointers are incremented with each reference. D1 initially contains the number of longwords (4 bytes) to be compared.

a. The initial contents of the registers are A0 = $00004000, A1 = $00005000 and D1 = $000000FF (the $ indicates hexadecimal notation). Memory between $4000 and $6000 is loaded with words $AAAA. If the foregoing program is run, specify the number of times the DBNE loop is executed and the contents of the three registers when the NOP instruction is reached.

b. Repeat (a), but now assume that memory between $4000 and $4FEE is loaded with $0000 and between $5000 and $6000 is loaded with $AAA.

14.15 Redraw Figures 14.19c, assuming that the conditional branch is not taken.

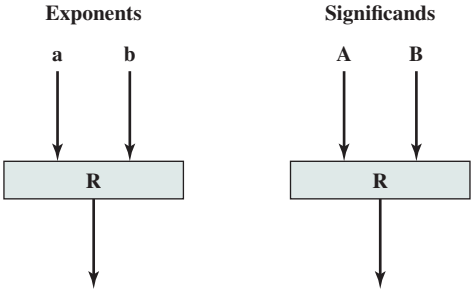
14.16 Table 14.5 summarizes statistics from [MACD84] concerning branch behavior for various classes of applications. With the exception of type 1 branch behavior, there is no noticeable difference among the application classes. Determine the fraction of all branches that go to the branch target address for the scientific environment. Repeat for commercial and systems environments.

14.17 Pipelining can be applied within the ALU to speed up floating-point operations. Consider the case of floating-point addition and subtraction. In simplified terms, the pipeline could have four stages: (1) Compare the exponents; (2) Choose the exponent and align the significands; (3) Add or subtract significands; (4) Normalize the results. The

Table 14.5 Branch Behavior in Sample Applications

Occurrence of branch classes:				
Type 1: Branch	72.5%			
Type 2: Loop control	9.8%			
Type 3: Procedure call, return	17.7%			
Type 1 branch: where it goes		**Scientific**	**Commercial**	**Systems**
Unconditional—100% go to target		20%	40%	35%
Conditional—went to target		43.2%	24.3%	32.5%
Conditional—did not go to target (inline)		36.8%	35.7%	32.5%
Type 2 branch (all environments)				
That go to target		91%		
That go inline		9%		
Type 3 branch				
100% go to target				

pipeline can be considered to have two parallel threads, one handling exponents and one handling significands, and could start out like this:



In this figure, the boxes labeled R refer to a set of registers used to hold temporary results. Complete the block diagram that shows at a top level the structure of the pipeline.

CHAPTER 15

Reduced Instruction Set Computers

15.1 Instruction Execution Characteristics
- Operations
- Operands
- Procedure Calls
- Implications

15.2 The Use of a Large Register File
- Register Windows
- Global Variables
- Large Register File versus Cache

15.3 Compiler-Based Register Optimization

15.4 Reduced Instruction Set Architecture
- Why CISC
- Characteristics of Reduced Instruction Set Architectures
- CISC versus RISC Characteristics

15.5 RISC Pipelining
- Pipelining with Regular Instructions
- Optimization of Pipelining

15.6 MIPS R4000
- Instruction Set
- Instruction Pipeline

15.7 SPARC
- SPARC Register Set
- Instruction Set
- Instruction Format

15.8 RISC versus CISC Controversy

15.9 Key Terms, Review Questions, and Problems

Learning Objectives

After studying this chapter, you should be able to:

- Provide an overview research results on instruction execution characteristics that motivated the development of the RISC approach.
- Summarize the key characteristics of RISC machines.
- Understand the design and performance implications of using a large register file.
- Understand the use of compiler-based register optimization to improve performance.
- Discuss the implication of a RISC architecture for pipeline design and performance.
- List and explain key approaches to pipeline optimization on a RISC machine.

Since the development of the stored-program computer around 1950, there have been remarkably few true innovations in the areas of computer organization and architecture. The following are some of the major advances since the birth of the computer:

- **The family concept:** Introduced by IBM with its System/360 in 1964, followed shortly thereafter by DEC, with its PDP-8. The family concept decouples the architecture of a machine from its implementation. A set of computers is offered, with different price/performance characteristics, that presents the same architecture to the user. The differences in price and performance are due to different implementations of the same architecture.
- **Microprogrammed control unit:** Suggested by Wilkes in 1951 and introduced by IBM on the S/360 line in 1964. Microprogramming eases the task of designing and implementing the control unit and provides support for the family concept.
- **Cache memory:** First introduced commercially on IBM S/360 Model 85 in 1968. The insertion of this element into the memory hierarchy dramatically improves performance.
- **Pipelining:** A means of introducing parallelism into the essentially sequential nature of a machine-instruction program. Examples are instruction pipelining and vector processing.
- **Multiple processors:** This category covers a number of different organizations and objectives.
- **Reduced instruction set computer (RISC) architecture:** This is the focus of this chapter.

When it appeared, RISC architecture was a dramatic departure from the historical trend in processor architecture. An analysis of the RISC architecture brings into focus many of the important issues in computer organization and architecture.

Although RISC architectures have been defined and designed in a variety of ways by different groups, the key elements shared by most designs are these:

- A large number of general-purpose registers, and/or the use of compiler technology to optimize register usage.
- A limited and simple instruction set.
- An emphasis on optimizing the instruction pipeline.

Table 15.1 compares several RISC and non-RISC systems.

We begin this chapter with a brief survey of some results on instruction sets, and then examine each of the three topics just listed. This is followed by a description of two of the best-documented RISC designs.

15.1 INSTRUCTION EXECUTION CHARACTERISTICS

One of the most visible forms of evolution associated with computers is that of programming languages. As the cost of hardware has dropped, the relative cost of software has risen. Along with that, a chronic shortage of programmers has driven up software costs in absolute terms. Thus, the major cost in the life cycle of a system is software, not hardware. Adding to the cost, and to the inconvenience, is the element of unreliability: it is common for programs, both system and application, to continue to exhibit new bugs after years of operation.

The response from researchers and industry has been to develop ever more powerful and complex high-level programming languages. These **high-level languages (HLLs)**: (1) allow the programmer to express algorithms more concisely; (2) allow the compiler to take care of details that are not important in the programmer's expression of algorithms; and (3) often support naturally the use of structured programming and/or object-oriented design.

Alas, this solution gave rise to a perceived problem, known as the *semantic gap*, the difference between the operations provided in HLLs and those provided in computer architecture. Symptoms of this gap are alleged to include execution inefficiency, excessive machine program size, and compiler complexity. Designers responded with architectures intended to close this gap. Key features include large instruction sets, dozens of addressing modes, and various HLL statements implemented in hardware. An example of the latter is the CASE machine instruction on the VAX. Such complex instruction sets are intended to:

- Ease the task of the compiler writer.
- Improve execution efficiency, because complex sequences of operations can be implemented in microcode.
- Provide support for even more complex and sophisticated HLLs.

Meanwhile, a number of studies have been done over the years to determine the characteristics and patterns of execution of machine instructions generated from HLL programs. The results of these studies inspired some researchers to look

Table 15.1 Characteristics of Some CISCs, RISCs, and Superscalar Processors

	Complex Instruction Set (CISC)Computer			Reduced Instruction Set (RISC) Computer	
Characteristic	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000
Year developed	1973	1978	1989	1987	1991
Number of instructions	208	303	235	69	94
Instruction size (bytes)	2–6	2–57	1–11	4	4
Addressing modes	4	22	11	1	1
Number of general-purpose registers	16	16	8	40–520	32
Control memory size (kbits)	420	480	246	—	—
Cache size (kB)	64	64	8	32	128

	Superscalar		
Characteristic	PowerPC	Ultra SPARC	MIPS R10000
Year developed	1993	1996	1996
Number of instructions	225		
Instruction size (bytes)	4	4	4
Addressing modes	2	1	1
Number of general-purpose registers	32	40–520	32
Control memory size (kbits)	—	—	—
Cache size (kB)	16–32	32	64

for a different approach: namely, to make the architecture that supports the HLL simpler, rather than more complex.

To understand the line of reasoning of the RISC advocates, we begin with a brief review of instruction execution characteristics. The aspects of computation of interest are as follows:

- **Operations performed:** These determine the functions to be performed by the processor and its interaction with memory.
- **Operands used:** The types of operands and the frequency of their use determine the memory organization for storing them and the addressing modes for accessing them.
- **Execution sequencing:** This determines the control and pipeline organization.

In the remainder of this section, we summarize the results of a number of studies of high-level-language programs. All of the results are based on dynamic measurements. That is, measurements are collected by executing the program and counting the number of times some feature has appeared or a particular property has held true. In contrast, static measurements merely perform these counts on the source text of a program. They give no useful information on performance, because they are not weighted relative to the number of times each statement is executed.

Operations

A variety of studies have been made to analyze the behavior of HLL programs. Table 4.7, discussed in Chapter 4, includes key results from a number of studies. There is quite good agreement in the results of this mixture of languages and applications. Assignment statements predominate, suggesting that the simple movement of data is of high importance. There is also a preponderance of conditional statements (IF, LOOP). These statements are implemented in machine language with some sort of compare and branch instruction. This suggests that the sequence control mechanism of the instruction set is important.

These results are instructive to the machine instruction set designer, indicating which types of statements occur most often and therefore should be supported in an "optimal" fashion. However, these results do not reveal which statements use the most time in the execution of a typical program. That is, we want to answer the question: Given a compiled machine-language program, which statements in the source language cause the execution of the most machine-language instructions and what is the execution time of these instructions?

To get at this underlying phenomenon, the Patterson programs [PATT82a], described in Appendix 4A, were compiled on the VAX, PDP-11, and Motorola 68000 to determine the average number of machine instructions and memory references per statement type. The second and third columns in Table 15.2 show the relative frequency of occurrence of various HLL statements in a variety of programs; the data were obtained by observing the occurrences in running programs rather than just the number of times that statements occur in the source code. Hence these metrics capture dynamic behavior. To obtain the data in columns four and five (machine-instruction weighted), each value in the second and third columns is multiplied by the number of machine instructions produced by the compiler. These results are then normalized so that columns four and five show the relative frequency of occurrence, weighted by the number of machine instructions per HLL statement. Similarly, the sixth and seventh columns are obtained by multiplying the frequency of occurrence of each statement type by the relative number of memory references caused by each statement. The data in columns four through seven provide surrogate measures of the actual time spent executing the various statement types. The results suggest that the procedure call/return is the most time-consuming operation in typical HLL programs.

The reader should be clear on the significance of Table 15.2. This table indicates the relative performance impact of various statement types in an HLL, when

Table 15.2 Weighted Relative Dynamic Frequency of HLL Operations [PATT82a]

	Dynamic Occurrence		Machine-Instruction Weighted		Memory-Reference Weighted	
	Pascal	C	Pascal	C	Pascal	C
ASSIGN	45%	38%	13%	13%	14%	15%
LOOP	5%	3%	42%	32%	33%	26%
CALL	15%	12%	31%	33%	44%	45%
IF	29%	43%	11%	21%	7%	13%
GOTO	—	3%	—	—	—	—
OTHER	6%	1%	3%	1%	2%	1%

that HLL is compiled for a typical contemporary instruction set architecture. Some other architecture could conceivably produce different results. However, this study produces results that are representative for contemporary **complex instruction set computer (CISC)** architectures. Thus, they can provide guidance to those looking for more efficient ways to support HLLs.

Operands

Much less work has been done on the occurrence of types of operands, despite the importance of this topic. There are several aspects that are significant.

The Patterson study already referenced [PATT82a] also looked at the dynamic frequency of occurrence of classes of variables (Table 15.3). The results, consistent between Pascal and C programs, show that most references are to simple scalar variables. Further, more than 80% of the scalars were local (to the procedure) variables. In addition, each reference to an array or a structure requires a reference to an index or pointer, which again is usually a local scalar. Thus, there is a preponderance of references to scalars, and these are highly localized.

The Patterson study examined the dynamic behavior of HLL programs, independent of the underlying architecture. As discussed before, it is necessary to deal with actual architectures to examine program behavior more deeply. One study, [LUND77], examined DEC-10 instructions dynamically and found that each instruction on the average references 0.5 operand in memory and 1.4 registers. Similar results are reported in [HUCK83] for C, Pascal, and FORTRAN programs on S/370, PDP-11, and VAX. Of course, these figures depend highly on both the architecture and the compiler, but they do illustrate the frequency of operand accessing.

Table 15.3 Dynamic Percentage of Operands

	Pascal	C	Average
Integer constant	16%	23%	20%
Scalar variable	58%	53%	55%
Array/Structure	26%	24%	25%

These latter studies suggest the importance of an architecture that lends itself to fast operand accessing, because this operation is performed so frequently. The Patterson study suggests that a prime candidate for optimization is the mechanism for storing and accessing local scalar variables.

Procedure Calls

We have seen that procedure calls and returns are an important aspect of HLL programs. The evidence (Table 15.2) suggests that these are the most time-consuming operations in compiled HLL programs. Thus, it will be profitable to consider ways of implementing these operations efficiently. Two aspects are significant: the number of parameters and variables that a procedure deals with, and the depth of nesting.

Tanenbaum's study [TANE78] found that 98% of dynamically called procedures were passed fewer than six arguments and that 92% of them used fewer than six local scalar variables. Similar results were reported by the Berkeley RISC team [KATE83], as shown in Table 15.4. These results show that the number of words required per procedure activation is not large. The studies reported earlier indicated that a high proportion of operand references is to local scalar variables. These studies show that those references are in fact confined to relatively few variables.

The same Berkeley group also looked at the pattern of procedure calls and returns in HLL programs. They found that it is rare to have a long uninterrupted sequence of procedure calls followed by the corresponding sequence of returns. Rather, they found that a program remains confined to a rather narrow window of procedure-invocation depth. This is illustrated in Figure 4.21, which was discussed in Chapter 4. These results reinforce the conclusion that operand references are highly localized.

Implications

A number of groups have looked at results such as those just reported and have concluded that the attempt to make the instruction set architecture close to HLLs is not the most effective design strategy. Rather, the HLLs can best be supported by optimizing performance of the most time-consuming features of typical HLL programs.

Table 15.4 Procedure Arguments and Local Scalar Variables

Percentage of Executed Procedure Calls With	Compiler, Interpreter, and Typesetter	Small Nonnumeric Programs
> 3 arguments	0–7%	0–5%
> 5 arguments	0–3%	0%
> 8 words of arguments and local scalars	1–20%	0–6%
> 12 words of arguments and local scalars	1–6%	0–3%

Generalizing from the work of a number of researchers, three elements emerge that, by and large, characterize RISC architectures. First, use a large number of registers or use a compiler to optimize register usage. This is intended to optimize operand referencing. The studies just discussed show that there are several references per HLL statement and that there is a high proportion of move (assignment) statements. This, coupled with the locality and predominance of scalar references, suggests that performance can be improved by reducing memory references at the expense of more register references. Because of the locality of these references, an expanded register set seems practical.

Second, careful attention needs to be paid to the design of instruction pipelines. Because of the high proportion of conditional branch and procedure call instructions, a straightforward instruction pipeline will be inefficient. This manifests itself as a high proportion of instructions that are prefetched but never executed.

Finally, an instruction set consisting of high-performance primitives is indicated. Instructions should have predictable costs (measured in execution time, code size, and increasingly, in energy dissipation) and be consistent with a high-performance implementation (which harmonizes with predictable execution-time cost).

15.2 THE USE OF A LARGE REGISTER FILE

The results summarized in Section 15.1 point out the desirability of quick access to operands. We have seen that there is a large proportion of assignment statements in HLL programs, and many of these are of the simple form A ← B. Also, there is a significant number of operand accesses per HLL statement. If we couple these results with the fact that most accesses are to local scalars, heavy reliance on register storage is suggested.

The reason that register storage is indicated is that it is the fastest available storage device, faster than both main memory and cache. The register file is physically small, on the same chip as the ALU and control unit, and employs much shorter addresses than addresses for cache and memory. Thus, a strategy is needed that will allow the most frequently accessed operands to be kept in registers and to minimize register-memory operations.

Two basic approaches are possible, one based on software and the other on hardware. The software approach is to rely on the compiler to maximize register usage. The compiler will attempt to assign registers to those variables that will be used the most in a given time period. This approach requires the use of sophisticated program-analysis algorithms. The hardware approach is simply to use more registers so that more variables can be held in registers for longer periods of time.

In this section, we will discuss the hardware approach. This approach has been pioneered by the Berkeley RISC group [PATT82a]; was used in the first commercial RISC product, the Pyramid [RAGA83]; and is currently used in the popular **SPARC** architecture.

Register Windows

On the face of it, the use of a large set of registers should decrease the need to access memory. The design task is to organize the registers in such a fashion that this goal is realized.

Because most operand references are to local scalars, the obvious approach is to store these in registers, with perhaps a few registers reserved for global variables. The problem is that the definition of *local* changes with each procedure call and return, operations that occur frequently. On every call, local variables must be saved from the registers into memory, so that the registers can be reused by the called procedure. Furthermore, parameters must be passed. On return, the variables of the calling procedure must be restored (loaded back into registers) and results must be passed back to the calling procedure.

The solution is based on two other results reported in Section 15.1. First, a typical procedure employs only a few passed parameters and local variables (Table 15.4). Second, the depth of procedure activation fluctuates within a relatively narrow range (Figure 4.21). To exploit these properties, multiple small sets of registers are used, each assigned to a different procedure. A procedure call automatically switches the processor to use a different fixed-size window of registers, rather than saving registers in memory. Windows for adjacent procedures are overlapped to allow parameter passing.

The concept is illustrated in Figure 15.1. At any time, only one window of registers is visible and is addressable as if it were the only set of registers (e.g., addresses 0 through $N - 1$). The window is divided into three fixed-size areas. Parameter registers hold parameters passed down from the procedure that called the current procedure and hold results to be passed back up. Local registers are used for local variables, as assigned by the compiler. Temporary registers are used to exchange parameters and results with the next lower level (procedure called by current procedure). The temporary registers at one level are physically the same as the parameter registers at the next lower level. This overlap permits parameters to be passed without the actual movement of data. Keep in mind that, except for the overlap, the registers at two different levels are physically distinct. That is, the parameter and local registers at level J are disjoint from the local and temporary registers at level $J + 1$.

To handle any possible pattern of calls and returns, the number of **register windows** would have to be unbounded. Instead, the register windows can be used to hold the few most recent procedure activations. Older activations must be saved

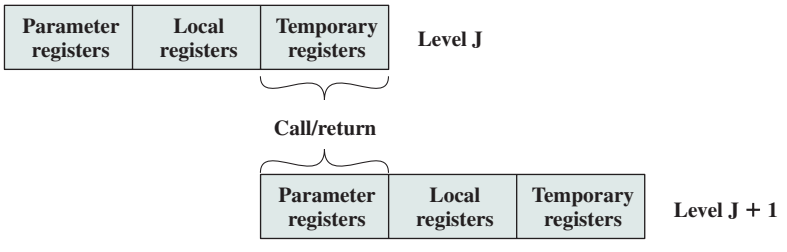


Figure 15.1 Overlapping Register Windows

in memory and later restored when the nesting depth decreases. Thus, the actual organization of the register file is as a circular buffer of overlapping windows. Two notable examples of this approach are Sun's SPARC architecture, described in Section 15.7, and the IA-64 architecture used in Intel's Itanium processor.

The circular organization is shown in Figure 15.2, which depicts a circular buffer of six windows. The buffer is filled to a depth of 4 (A called B; B called C; C called D) with procedure D active. The current-window pointer (CWP) points to the window of the currently active procedure. Register references by a machine instruction are offset by this pointer to determine the actual physical register. The saved-window pointer (SWP) identifies the window most recently saved in memory. If procedure D now calls procedure E, arguments for E are placed in D's temporary registers (the overlap between w3 and w4) and the CWP is advanced by one window.

If procedure E then makes a call to procedure F, the call cannot be made with the current status of the buffer. This is because F's window overlaps A's window. If F begins to load its temporary registers, preparatory to a call, it will overwrite the parameter registers of A (A.in). Thus, when CWP is incremented (modulo 6) so that it becomes equal to SWP, an interrupt occurs, and A's window is saved. Only

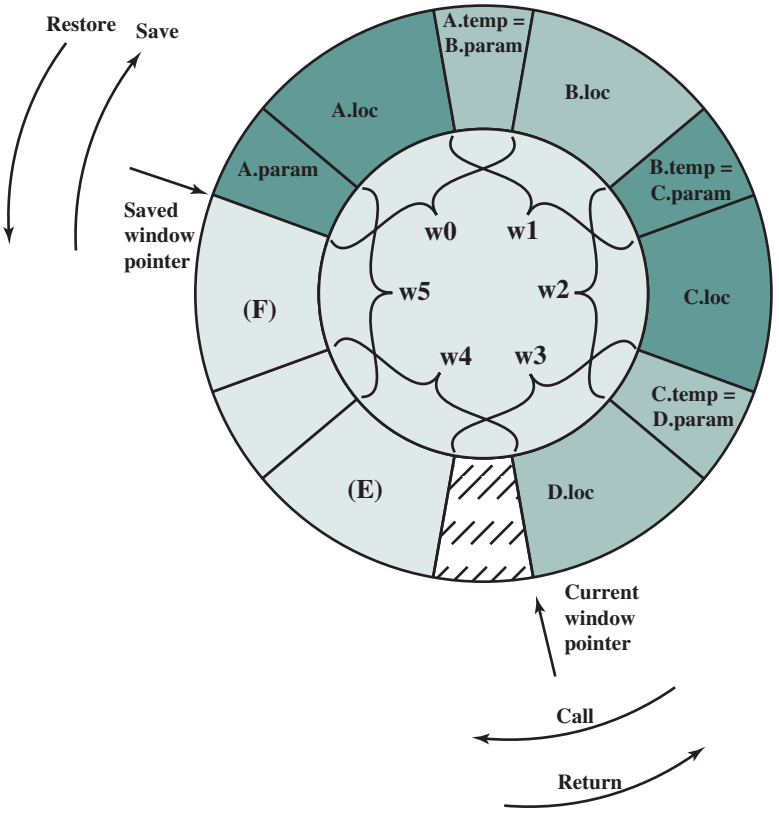


Figure 15.2 Circular-Buffer Organization of Overlapped Windows

the first two portions (A.in and A.loc) need be saved. Then, the SWP is incremented and the call to F proceeds. A similar interrupt can occur on returns. For example, subsequent to the activation of F, when B returns to A, CWP is decremented and becomes equal to SWP. This causes an interrupt that results in the restoration of A's window.

From the preceding, it can be seen that an N-window register file can hold only $N - 1$ procedure activations. The value of N need not be large. As was mentioned in Appendix 4A, one study [TAMI83] found that, with 8 windows, a save or restore is needed on only 1% of the calls or returns. The Berkeley RISC computers use 8 windows of 16 registers each. The Pyramid computer employs 16 windows of 32 registers each.

Global Variables

The window scheme just described provides an efficient organization for storing local scalar variables in registers. However, this scheme does not address the need to store global variables, those accessed by more than one procedure. Two options suggest themselves. First, variables declared as global in an HLL can be assigned memory locations by the compiler, and all machine instructions that reference these variables will use memory-reference operands. This is straightforward, from both the hardware and software (compiler) points of view. However, for frequently accessed global variables, this scheme is inefficient.

An alternative is to incorporate a set of global registers in the processor. These registers would be fixed in number and available to all procedures. A unified numbering scheme can be used to simplify the instruction format. For example, references to registers 0 through 7 could refer to unique global registers, and references to registers 8 through 31 could be offset to refer to physical registers in the current window. There is an increased hardware burden to accommodate the split in register addressing. In addition, the linker must decide which global variables should be assigned to registers.

Large Register File versus Cache

The register file, organized into windows, acts as a small, fast buffer for holding a subset of all variables that are likely to be used the most heavily. From this point of view, the register file acts much like a cache memory, although a much faster memory. The question therefore arises as to whether it would be simpler and better to use a cache and a small traditional register file.

Table 15.5 compares characteristics of the two approaches. The window-based register file holds all the local scalar variables (except in the rare case of window overflow) of the most recent $N - 1$ procedure activations. The cache holds a selection of recently used scalar variables. The register file should save time, because all local scalar variables are retained. On the other hand, the cache may make more efficient use of space, because it is reacting to the situation dynamically. Furthermore, caches generally treat all memory references alike, including instructions and other types of data. Thus, savings in these other areas are possible with a cache and not a register file.

Table 15.5 Characteristics of Large-Register-File and Cache Organizations

Large Register File	Cache
All local scalars	Recently-used local scalars
Individual variables	Blocks of memory
Compiler-assigned global variables	Recently-used global variables
Save/Restore based on procedure nesting depth	Save/Restore based on cache replacement algorithm
Register addressing	Memory addressing
Multiple operands addressed and accessed in one cycle	One operand addressed and accessed per cycle

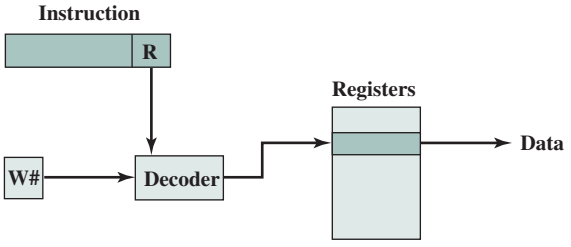
A register file may make inefficient use of space, because not all procedures will need the full window space allotted to them. On the other hand, the cache suffers from another sort of inefficiency: Data are read into the cache in blocks. Whereas the register file contains only those variables in use, the cache reads in a block of data, some or much of which will not be used.

The cache is capable of handling global as well as local variables. There are usually many global scalars, but only a few of them are heavily used [KATE83]. A cache will dynamically discover these variables and hold them. If the window-based register file is supplemented with global registers, it too can hold some global scalars. However, when program modules are separately compiled, it is impossible for the compiler to assign global values to registers; the linker must perform this task.

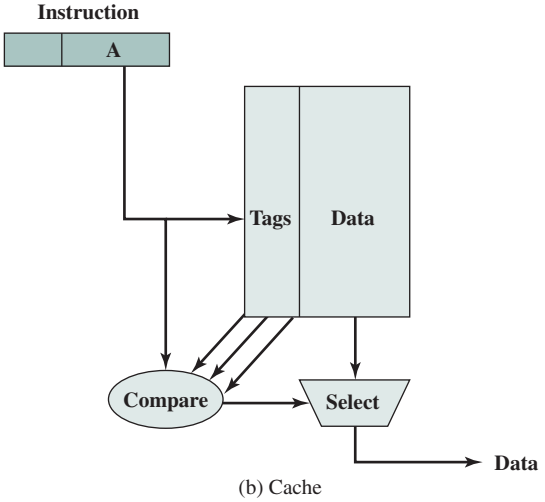
With the register file, the movement of data between registers and memory is determined by the procedure nesting depth. Because this depth usually fluctuates within a narrow range, the use of memory is relatively infrequent. Most cache memories are set associative with a small set size. Thus, there is the danger that other data or instructions will compete for cache residency.

Based on the discussion so far, the choice between a large window-based register file and a cache is not clear-cut. There is one characteristic, however, in which the register approach is clearly superior and which suggests that a cache-based system will be noticeably slower. This distinction shows up in the amount of addressing overhead experienced by the two approaches.

Figure 15.3 illustrates the difference. To reference a local scalar in a window-based register file, a “virtual” register number and a window number are used. These can pass through a relatively simple decoder to select one of the physical registers. To reference a memory location in cache, a full-width memory address must be generated. The complexity of this operation depends on the addressing mode. In a set associative cache, a portion of the address is used to read a number of words and tags equal to the set size. Another portion of the address is compared with the tags, and one of the words that were read is selected. It should be clear that even if the cache is as fast as the register file, the access time will be considerably longer. Thus, from the point of view of performance, the window-based register file is superior for local scalars. Further performance improvement could be achieved by the addition of a cache for instructions only.



(a) Window-based register file



(b) Cache

Figure 15.3 Referencing a Scalar

15.3 COMPILER-BASED REGISTER OPTIMIZATION

Let us assume now that only a small number (e.g., 16–32) of registers is available on the target RISC machine. In this case, optimized register usage is the responsibility of the compiler. A program written in a high-level language has, of course, no explicit references to registers (the C-language keyword register notwithstanding). Rather, program quantities are referred to symbolically. The objective of the compiler is to keep the operands for as many computations as possible in registers rather than main memory, and to minimize load-and-store operations.

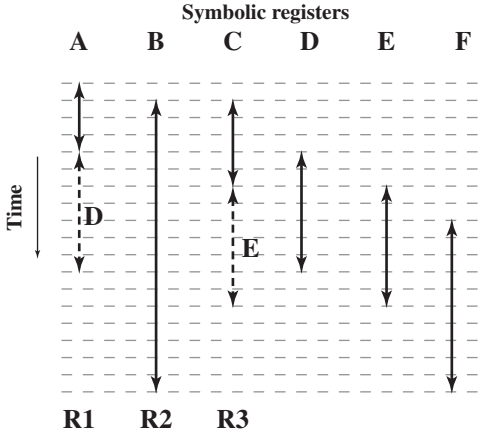
In general, the approach taken is as follows. Each program quantity that is a candidate for residing in a register is assigned to a symbolic or virtual register. The compiler then maps the unlimited number of symbolic registers into a fixed number of real registers. Symbolic registers whose usage does not overlap can share the same real register. If, in a particular portion of the program, there are more quantities to deal with than real registers, then some of the quantities are assigned to memory locations. Load-and-store instructions are used to position quantities in registers temporarily for computational operations.

The essence of the optimization task is to decide which quantities are to be assigned to registers at any given point in the program. The technique most commonly used in RISC compilers is known as graph coloring, which is a technique borrowed from the discipline of topology [CHAI82, CHOW86, COUT86, CHOW90].

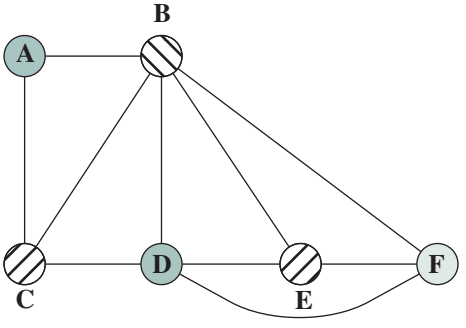
The graph coloring problem is this. Given a graph consisting of nodes and edges, assign colors to nodes such that adjacent nodes have different colors, and do this in such a way as to minimize the number of different colors. This problem is adapted to the compiler problem in the following way. First, the program is analyzed to build a register interference graph. The nodes of the graph are the symbolic registers. If two symbolic registers are "live" during the same program fragment, then they are joined by an edge to depict interference. An attempt is then made to color the graph with n colors, where n is the number of registers. Nodes that share the same color can be assigned to the same register. If this process does not fully succeed, then those nodes that cannot be colored must be placed in memory, and loads and stores must be used to make space for the affected quantities when they are needed.

Figure 15.4 is a simple example of the process. Assume a program with six symbolic registers to be compiled into three actual registers. Figure 15.4a shows the time sequence of active use of each symbolic register. The dashed horizontal lines indicate successive instruction executions. Figure 15.4b shows the register interference graph (shading and stripes are used instead of colors). A possible coloring with three colors is indicated. Because symbolic registers A and D do not interfere, the compile can assign both of these to physical register R1. Similarly, symbolic registers C and E can be assigned to register R3. One symbolic register, F, is left uncolored and must be dealt with using loads and stores.

In general, there is a trade-off between the use of a large set of registers and compiler-based register optimization. For example, [BRAD91a] reports on a study



(a) Time sequence of active use of registers



(b) Register interference graph

Figure 15.4 Graph Coloring Approach

that modeled a RISC architecture with features similar to the Motorola 88000 and the MIPS R2000. The researchers varied the number of registers from 16 to 128, and they considered both the use of all general-purpose registers and registers split between integer and floating-point use. Their study showed that with even simple register optimization, there is little benefit to the use of more than 64 registers. With reasonably sophisticated register optimization techniques, there is only marginal performance improvement with more than 32 registers. Finally, they noted that with a small number of registers (e.g., 16), a machine with a shared register organization executes faster than one with a split organization. Similar conclusions can be drawn from [HUGU91], which reports on a study that is primarily concerned with optimizing the use of a small number of registers rather than comparing the use of large register sets with optimization efforts.

15.4 REDUCED INSTRUCTION SET ARCHITECTURE

In this section, we look at some of the general characteristics of and the motivation for a reduced instruction set architecture. Specific examples will be seen later in this chapter. We begin with a discussion of motivations for contemporary complex instruction set architectures.

Why CISC

We have noted the trend to richer instruction sets, which include a larger number of instructions and more complex instructions. Two principal reasons have motivated this trend: a desire to simplify compilers and a desire to improve performance. Underlying both of these reasons was the shift to HLLs on the part of programmers; architects attempted to design machines that provided better support for HLLs.

It is not the intent of this chapter to say that the CISC designers took the wrong direction. Indeed, because technology continues to evolve and because architectures exist along a spectrum rather than in two neat categories, a black-and-white assessment is unlikely ever to emerge. Thus, the comments that follow are simply meant to point out some of the potential pitfalls in the CISC approach and to provide some understanding of the motivation of the RISC adherents.

The first of the reasons cited, compiler simplification, seems obvious, but it is not. The task of the compiler writer is to build a compiler that generates good (fast, small, fast and small) sequences of machine instructions for HLL programs (i.e., the compiler views individual HLL statements in the context of surrounding HLL statements). If there are machine instructions that resemble HLL statements, this task is simplified. This reasoning has been disputed by the RISC researchers ([HENN82], [RADI83], [PATT82b]). They have found that complex machine instructions are often hard to exploit because the compiler must find those cases that exactly fit the construct. The task of optimizing the generated code to minimize code size, reduce instruction execution count, and enhance pipelining is much more difficult with a complex instruction set. As evidence of this, studies cited earlier in this chapter indicate that most of the instructions in a compiled program are the relatively simple ones.

The other major reason cited is the expectation that a CISC will yield smaller, faster programs. Let us examine both aspects of this assertion: that programs will be smaller and that they will execute faster.

There are two advantages to smaller programs. Because the program takes up less memory, there is a savings in that resource. With memory today being so inexpensive, this potential advantage is no longer compelling. More important, smaller programs should improve performance, and this will happen in three ways. First, fewer instructions means fewer instruction bytes to be fetched. Second, in a paging environment, smaller programs occupy fewer pages, reducing page faults. Third, more instructions fit in cache(s).

The problem with this line of reasoning is that it is far from certain that a CISC program will be smaller than a corresponding RISC program. In many cases, the CISC program, expressed in symbolic machine language, may be *shorter* (i.e., fewer instructions), but the number of bits of memory occupied may not be noticeably *smaller.* Table 15.6 shows results from three studies that compared the size of compiled C programs on a variety of machines, including RISC I, which has a reduced instruction set architecture. Note that there is little or no savings using a CISC over a RISC. It is also interesting to note that the VAX, which has a much more complex instruction set than the PDP-11, achieves very little savings over the latter. These results were confirmed by IBM researchers [RADI83], who found that the IBM 801 (a RISC) produced code that was 0.9 times the size of code on an IBM S/370. The study used a set of PL/I programs.

There are several reasons for these rather surprising results. We have already noted that compilers on CISCs tend to favor simpler instructions, so that the conciseness of the complex instructions seldom comes into play. Also, because there are more instructions on a CISC, longer opcodes are required, producing longer instructions. Finally, RISCs tend to emphasize register rather than memory references, and the former require fewer bits. An example of this last effect is discussed presently.

So the expectation that a CISC will produce smaller programs, with the attendant advantages, may not be realized. The second motivating factor for increasingly complex instruction sets was that instruction execution would be faster. It seems to make sense that a complex HLL operation will execute more quickly as a single machine instruction rather than as a series of more primitive instructions. However, because of the bias toward the use of those simpler instructions, this may not be so.

Table 15.6 Code Size Relative to RISC I

	[PATT82a] 11 C Programs	[KATE83] 12 C Programs	[HEAT84] 5 C Programs
RISC I	1.0	1.0	1.0
VAX-11/780	0.8	0.67	
M68000	0.9		0.9
Z8002	1.2		1.12
PDP-11/70	0.9	0.71	

The entire control unit must be made more complex, and/or the microprogram control store must be made larger, to accommodate a richer instruction set. Either factor increases the execution time of the simple instructions.

In fact, some researchers have found that the speedup in the execution of complex functions is due not so much to the power of the complex machine instructions as to their residence in high-speed control store [RADI83]. In effect, the control store acts as an instruction cache. Thus, the hardware architect is in the position of trying to determine which subroutines or functions will be used most frequently and assigning those to the control store by implementing them in microcode. The results have been less than encouraging. On S/390 systems, instructions such as Translate and Extended-Precision-Floating-Point-Divide reside in high-speed storage, while the sequence involved in setting up procedure calls or initiating an interrupt handler are in slower main memory.

Thus, it is far from clear that a trend to increasingly complex instruction sets is appropriate. This has led a number of groups to pursue the opposite path.

Characteristics of Reduced Instruction Set Architectures

Although a variety of different approaches to reduced instruction set architecture have been taken, certain characteristics are common to all of them:

- One instruction per cycle
- Register-to-register operations
- Simple addressing modes
- Simple instruction formats

Here, we provide a brief discussion of these characteristics. Specific examples are explored later in this chapter.

The first characteristic listed is that there is **one machine instruction per machine cycle**. A *machine cycle* is defined to be the time it takes to fetch two operands from registers, perform an ALU operation, and store the result in a register. Thus, RISC machine instructions should be no more complicated than, and execute about as fast as, microinstructions on CISC machines (discussed in Part Four). With simple, one-cycle instructions, there is little or no need for microcode; the machine instructions can be hardwired. Such instructions should execute faster than comparable machine instructions on other machines, because it is not necessary to access a microprogram control store during instruction execution.

A second characteristic is that most operations should be **register to register**, with only simple LOAD and STORE operations accessing memory. This design feature simplifies the instruction set and therefore the control unit. For example, a RISC instruction set may include only one or two ADD instructions (e.g., integer add, add with carry); the VAX has 25 different ADD instructions. Another benefit is that such an architecture encourages the optimization of register use, so that frequently accessed operands remain in high-speed storage.

This emphasis on register-to-register operations is notable for RISC designs. Contemporary CISC machines provide such instructions but also include memory-to-memory and mixed register/memory operations. Attempts to compare these

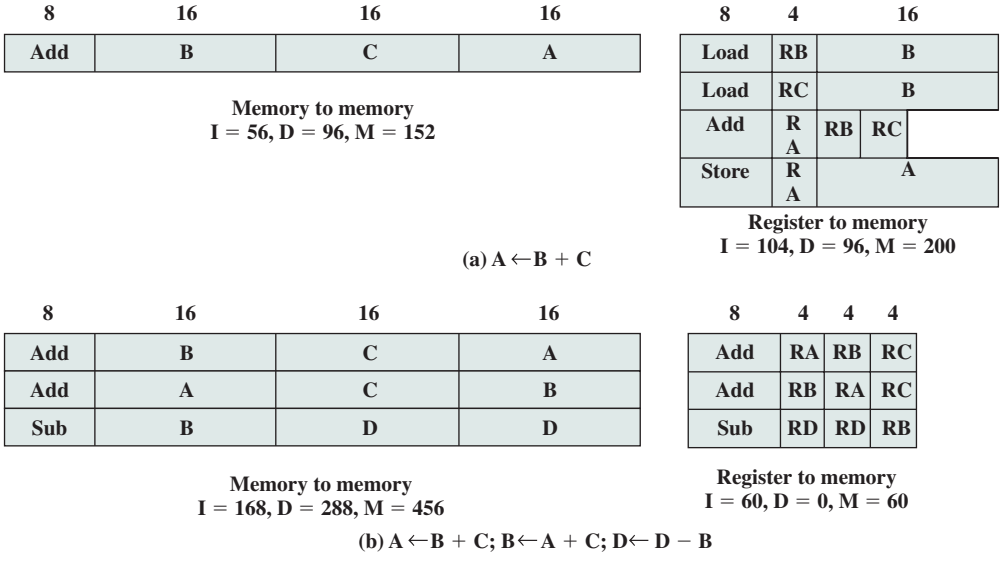

8 16 16 16
Add B C A
Memory to memory
I = 56, D = 96, M = 152
8 4 16
Load RB B
Load RC B
Add RA RB RC
Store RA A
Register to memory
I = 104, D = 96, M = 200
(a) A ← B + C
8 16 16 16
Add B C A
Add A C B
Sub B D D
Memory to memory
I = 168, D = 288, M = 456
8 4 4 4
Add RA RB RC
Add RB RA RC
Sub RD RD RB
Register to memory
I = 60, D = 0, M = 60
(b) A ← B + C; B ← A + C; D ← D − B
I = number of bytes occupied by executed instructions
D = number of bytes occupied by data
M = total memory traffic = I + D

Figure 15.5 Two Comparisons of Register-to-Register and Memory-to-Memory Approaches

approaches were made in the 1970s, before the appearance of RISCs. Figure 15.5a illustrates the approach taken. Hypothetical architectures were evaluated on program size and the number of bits of memory traffic. Results such as this one led one researcher to suggest that future architectures should contain no registers at all [MYER78]. One wonders what he would have thought, at the time, of the RISC machine once produced by Pyramid, which contained no less than 528 registers!

What was missing from those studies was a recognition of the frequent access to a small number of local scalars and that, with a large bank of registers or an optimizing compiler, most operands could be kept in registers for long periods of time. Thus, Figure 15.5b may be a fairer comparison.

A third characteristic is the use of **simple addressing modes**. Almost all RISC instructions use simple register addressing. Several additional modes, such as displacement and PC-relative, may be included. Other, more complex modes can be synthesized in software from the simple ones. Again, this design feature simplifies the instruction set and the control unit.

A final common characteristic is the use of **simple instruction formats**. Generally, only one or a few formats are used. Instruction length is fixed and aligned on word boundaries. Field locations, especially the opcode, are fixed. This design feature has a number of benefits. With fixed fields, opcode decoding and register operand accessing can occur simultaneously. Simplified formats simplify the control unit. Instruction fetching is optimized because word-length units are fetched. Alignment on a word boundary also means that a single instruction does not cross page boundaries.

Taken together, these characteristics can be assessed to determine the potential performance benefits of the RISC approach. A certain amount of "circumstantial

evidence" can be presented. First, more effective optimizing compilers can be developed. With more-primitive instructions, there are more opportunities for moving functions out of loops, reorganizing code for efficiency, maximizing register utilization, and so forth. It is even possible to compute parts of complex instructions at compile time. For example, the S/390 Move Characters (MVC) instruction moves a string of characters from one location to another. Each time it is executed, the move will depend on the length of the string, whether and in which direction the locations overlap, and what the alignment characteristics are. In most cases, these will all be known at compile time. Thus, the compiler could produce an optimized sequence of primitive instructions for this function.

A second point, already noted, is that most instructions generated by a compiler are relatively simple anyway. It would seem reasonable that a control unit built specifically for those instructions and using little or no microcode could execute them faster than a comparable CISC.

A third point relates to the use of instruction pipelining. RISC researchers feel that the instruction pipelining technique can be applied much more effectively with a reduced instruction set. We examine this point in some detail presently.

A final, and somewhat less significant, point is that RISC processors are more responsive to interrupts because interrupts are checked between rather elementary operations. Architectures with complex instructions either restrict interrupts to instruction boundaries or must define specific interruptible points and implement mechanisms for restarting an instruction.

The case for improved performance for a reduced instruction set architecture is strong, but one could perhaps still make an argument for CISC. A number of studies have been done, but not on machines of comparable technology and power. Further, most studies have not attempted to separate the effects of a reduced instruction set and the effects of a large register file. The "circumstantial evidence," however, is suggestive.

CISC versus RISC Characteristics

After the initial enthusiasm for RISC machines, there has been a growing realization that (1) RISC designs may benefit from the inclusion of some CISC features and that (2) CISC designs may benefit from the inclusion of some RISC features. The result is that the more recent RISC designs, notably the PowerPC, are no longer "pure" RISC and the more recent CISC designs, notably the Pentium II and later Pentium models, do incorporate some RISC characteristics.

An interesting comparison in [MASH95] provides some insight into this issue. Table 15.7 lists a number of processors and compares them across a number of characteristics. For purposes of this comparison, the following are considered typical of a classic RISC:

1. A single instruction size.
2. That size is typically 4 bytes.
3. A small number of data addressing modes, typically less than five. This parameter is difficult to pin down. In the table, register and literal modes are not counted and different formats with different offset sizes are counted separately.

Table 15.7 Characteristics of Some Processors

Processor	Number of instruction sizes	Max instruction size in bytes	Number of addressing modes	Indirect addressing	Load/store combined with arithmetic	Max number of memory operands	Unaligned addressing allowed	Max number of MMU uses	Number of bits for integer register specifier	Number of bits for FP register specifier
AMD29000	1	4	1	no	no	1	no	1	8	3[a]
MIPS R2000	1	4	1	no	no	1	no	1	5	4
SPARC	1	4	2	no	no	1	no	1	5	4
MC88000	1	4	3	no	no	1	no	1	5	4
HP PA	1	4	10[a]	no	no	1	no	1	5	4
IBM RT/PC	2[a]	4	1	no	no	1	no	1	4[a]	3[a]
IBM RS/6000	1	4	4	no	no	1	yes	1	5	5
Intel i860	1	4	4	no	no	1	no	1	5	4
IBM 3090	4	8	2[b]	no[b]	yes	2	yes	4	4	2
Intel 80486	12	12	15	no[b]	yes	2	yes	4	3	3
NSC 32016	21	21	23	yes	yes	2	yes	4	3	3
MC68040	11	22	44	yes	yes	2	yes	8	4	3
VAX	56	56	22	yes	yes	6	yes	24	4	0
Clipper	4[a]	8[a]	9[a]	no	no	1	0	2	4[a]	3[a]
Intel 80960	2[a]	8[a]	9[a]	no	no	1	yes[a]	—	5	3[a]

Notes: [a] RISC that does not conform to this characteristic.
[b] CISC that does not conform to this characteristic.

4. No indirect addressing that requires you to make one memory access to get the address of another operand in memory.
5. No operations that combine load/store with arithmetic (e.g., add from memory, add to memory).
6. No more than one memory-addressed operand per instruction.
7. Does not support arbitrary alignment of data for load/store operations.
8. Maximum number of uses of the memory management unit (MMU) for a data address in an instruction.
9. Number of bits for integer register specifier equal to five or more. This means that at least 32 integer registers can be explicitly referenced at a time.
10. Number of bits for floating-point register specifier equal to four or more. This means that at least 16 floating-point registers can be explicitly referenced at a time.

Items 1 through 3 are an indication of instruction decode complexity. Items 4 through 8 suggest the ease or difficulty of pipelining, especially in the presence of virtual memory requirements. Items 9 and 10 are related to the ability to take good advantage of compilers.

In the table, the first eight processors are clearly RISC architectures, the next five are clearly CISC, and the last two are processors often thought of as RISC that in fact have many CISC characteristics.

15.5 RISC PIPELINING

Pipelining with Regular Instructions

As we discussed in Section 12.4, instruction pipelining is often used to enhance performance. Let us reconsider this in the context of a RISC architecture. Most instructions are register to register, and an instruction cycle has the following two stages:

- I: Instruction fetch.
- E: Execute. Performs an ALU operation with register input and output.

For load and store operations, three stages are required:

- I: Instruction fetch.
- E: Execute. Calculates memory address.
- D: Memory. Register-to-memory or memory-to-register operation.

Figure 15.6a depicts the timing of a sequence of instructions using no pipelining. Clearly, this is a wasteful process. Even very simple pipelining can substantially improve performance. Figure 15.6b shows a two-stage pipelining scheme, in which the I and E stages of two different instructions are performed simultaneously. The two stages of the pipeline are an instruction fetch stage, and an execute/memory stage that executes the instruction, including register-to-memory and memory-to-register operations. Thus we see that the instruction fetch stage of the

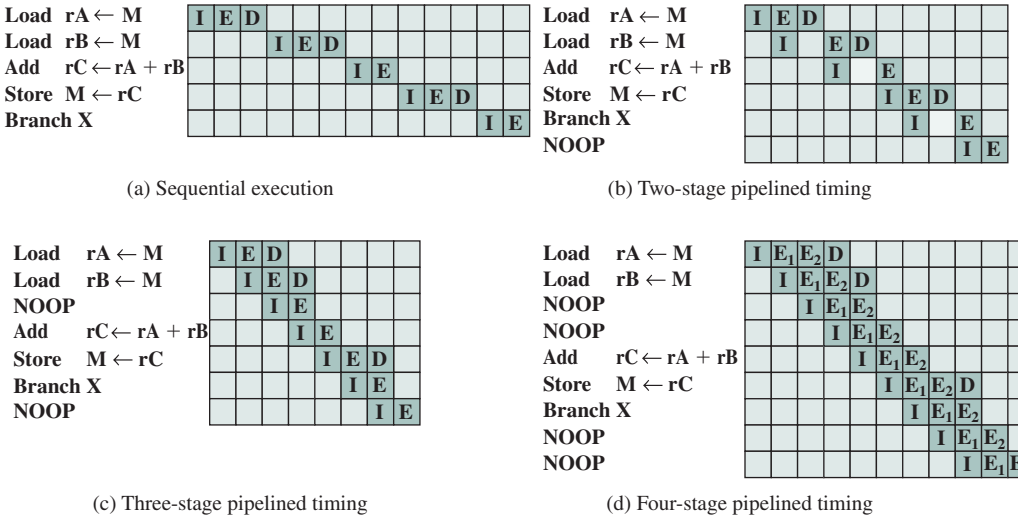


Figure 15.6 The Effects of Pipelining

second instruction can be performed in parallel with the first part of the execute/memory stage. However, the execute/memory stage of the second instruction must be delayed until the first instruction clears the second stage of the pipeline. This scheme can yield up to twice the execution rate of a serial scheme. Two problems prevent the maximum speedup from being achieved. First, we assume that a single-port memory is used and that only one memory access is possible per stage. This requires the insertion of a wait state in some instructions. Second, a branch instruction interrupts the sequential flow of execution. To accommodate this with minimum circuitry, a NOOP instruction can be inserted into the instruction stream by the compiler or assembler.

Pipelining can be improved further by permitting two memory accesses per stage. This yields the sequence shown in Figure 15.6c. Now, up to three instructions can be overlapped, and the improvement is as much as a factor of 3. Again, branch instructions cause the speedup to fall short of the maximum possible. Also, note that data dependencies have an effect. If an instruction needs an operand that is altered by the preceding instruction, a delay is required. Again, this can be accomplished by a NOOP.

The pipelining discussed so far works best if the three stages are of approximately equal duration. Because the E stage usually involves an ALU operation, it may be longer. In this case, we can divide into two substages:

- E_1: Register file read
- E_2: ALU operation and register write

Because of the simplicity and regularity of a RISC instruction set, the design of the phasing into three or four stages is easily accomplished. Figure 15.6d shows the result with a four-stage pipeline. Up to four instructions at a time can be under way, and the maximum potential speedup is a factor of 4. Note again the use of NOOPs to account for data and branch delays.

Optimization of Pipelining

Because of the simple and regular nature of RISC instructions, it is easier for a hardware designer to implement a simple, fast pipeline. There are few variations in instruction execution duration, and the pipeline can be tailored to reflect this. However, we have seen that data and branch dependencies reduce the overall execution rate.

DELAYED BRANCH To compensate for these dependencies, code reorganization techniques have been developed. First, let us consider branching instructions. **Delayed branch**, a way of increasing the efficiency of the pipeline, makes use of a branch that does not take effect until after execution of the following instruction (hence the term *delayed*). The instruction location immediately following the branch is referred to as the *delay slot.* This strange procedure is illustrated in Table 15.8. In the column labeled "normal branch," we see a normal symbolic instruction machine-language program. After 102 is executed, the next instruction to be executed is 105. To regularize the pipeline, a NOOP is inserted after this branch. However, increased performance is achieved if the instructions at 101 and 102 are interchanged.

Figure 15.7 shows the result. Figure 15.7a shows the traditional approach to pipelining, of the type discussed in Chapter 14 (e.g., see Figures 14.11 and 14.12). The JUMP instruction is fetched at time 4. At time 5, the JUMP instruction is executed at the same time that instruction 103 (ADD instruction) is fetched. Because a JUMP occurs, which updates the program counter, the pipeline must be cleared of instruction 103; at time 6, instruction 105, which is the target of the JUMP, is loaded. Figure 15.7b shows the same pipeline handled by a typical RISC organization. The timing is the same. However, because of the insertion of the NOOP instruction, we do not need special circuitry to clear the pipeline; the NOOP simply executes with no effect. Figure 15.7c shows the use of the delayed branch. The JUMP instruction is fetched at time 2, before the ADD instruction, which is fetched at time 3. Note, however, that the ADD instruction is fetched before the execution of the JUMP instruction has a chance to alter the program counter. Therefore, during time 4, the ADD instruction is executed at the same time that instruction 105 is fetched. Thus, the original semantics of the program are retained but two fewer clock cycles are required for execution.

This interchange of instructions will work successfully for unconditional branches, calls, and returns. For conditional branches, this procedure cannot be blindly applied. If the condition that is tested for the branch can be altered by the

Table 15.8 Normal and Delayed Branch

Address	Normal Branch		Delayed Branch		Optimized Delayed Branch	
100	LOAD	X, rA	LOAD	X, rA	LOAD	X, rA
101	ADD	1, rA	ADD	1, rA	JUMP	105
102	JUMP	105	JUMP	106	ADD	1, rA
103	ADD	rA, rB	NOOP		ADD	rA, rB
104	SUB	rC, rB	ADD	rA, rB	SUB	rC, rB
105	STORE	rA, Z	SUB	rC, rB	STORE	rA, Z
106			STORE	rA, Z		

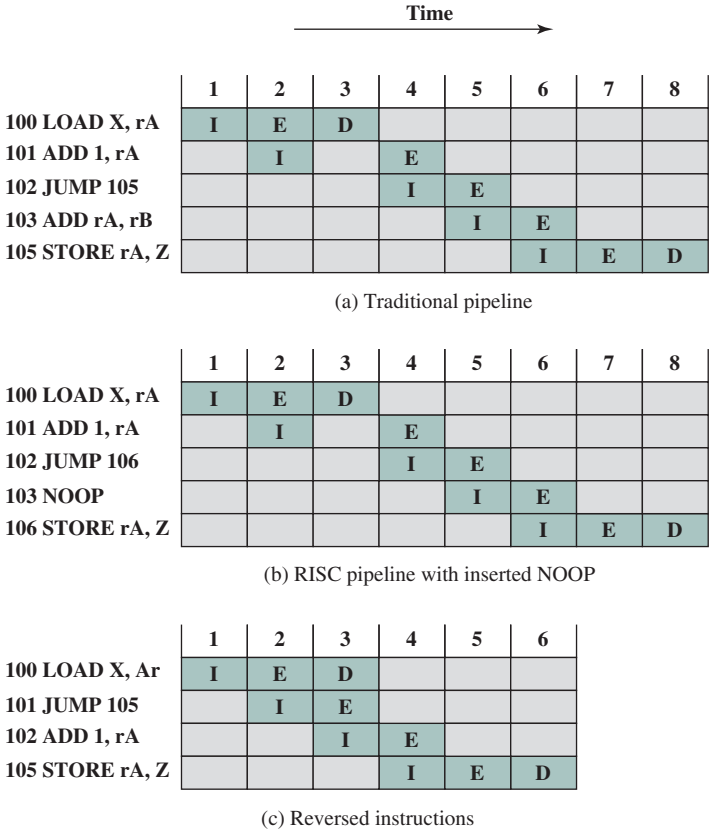


Figure 15.7 Use of the Delayed Branch

immediately preceding instruction, then the compiler must refrain from doing the interchange and instead insert a NOOP. Otherwise, the compiler can seek to insert a useful instruction after the branch. The experience with both the Berkeley RISC and IBM 801 systems is that the majority of conditional branch instructions can be optimized in this fashion ([PATT82a], [RADI83]).

DELAYED LOAD A similar sort of tactic, called the **delayed load**, can be used on LOAD instructions. On LOAD instructions, the register that is to be the target of the load is locked by the processor. The processor then continues execution of the instruction stream until it reaches an instruction requiring that register, at which point it idles until the load is complete. If the compiler can rearrange instructions so that useful work can be done while the load is in the pipeline, efficiency is increased.

Loop Unrolling Simulator

```
do i=2, n-1
        a[i] = a[i] + a[i-1] * a[i+1]
end do
```

(a) Original loop

```
do i=2, n-2, 2
        a[i] = a[i] + a[i-1] * a[i+1]
        a[i+1] = a[i+1] + a[i] * a[i+2]
end do

if (mod(n-2, 2) = i) then
    a[n-1] = a[n-1] + a[n-2] * a[n]
end if
```

(b) Loop unrolled twice

Figure 15.8 Loop Unrolling

LOOP UNROLLING Another compiler technique to improve instruction parallelism is loop unrolling [BACO94]. Unrolling replicates the body of a loop some number of times called the unrolling factor (u) and iterates by step u instead of step 1.

Unrolling can improve the performance by

- reducing loop overhead
- increasing instruction parallelism by improving pipeline performance
- improving register, data cache, or TLB locality

Figure 15.8 illustrates all three of these improvements in an example. Loop overhead is cut in half because two iterations are performed before the test and branch at the end of the loop. Instruction parallelism is increased because the second assignment can be performed while the results of the first are being stored and the loop variables are being updated. If array elements are assigned to registers, register locality will improve because a[i] and a[i + 1] are used twice in the loop body, reducing the number of loads per iteration from three to two.

As a final note, we should point out that the design of the instruction pipeline should not be carried out in isolation from other optimization techniques applied to the system. For example, [BRAD91b] shows that the scheduling of instructions for the pipeline and the dynamic allocation of registers should be considered together to achieve the greatest efficiency.

15.6 MIPS R4000

One of the first commercially available RISC chip sets was developed by MIPS Technology Inc. The system was inspired by an experimental system, also using the name MIPS, developed at Stanford [HENN84]. In this section we look at the MIPS

R4000. It has substantially the same architecture and instruction set of the earlier MIPS designs: the R2000 and R3000. The most significant difference is that the R4000 uses 64 rather than 32 bits for all internal and external data paths and for addresses, registers, and the ALU.

The use of 64 bits has a number of advantages over a 32-bit architecture. It allows a bigger address space—large enough for an operating system to map more than a terabyte of files directly into virtual memory for easy access. With 1-terabyte and larger disk drives now common, the 4-gigabyte address space of a 32-bit machine becomes limiting. Also, the 64-bit capacity allows the R4000 to process data such as IEEE double-precision floating-point numbers and character strings, up to eight characters in a single action.

The R4000 processor chip is partitioned into two sections, one containing the CPU and the other containing a coprocessor for memory management. The processor has a very simple architecture. The intent was to design a system in which the instruction execution logic was as simple as possible, leaving space available for logic to enhance performance (e.g., the entire memory-management unit).

The processor supports thirty-two 64-bit registers. It also provides for up to 128 Kbytes of high-speed cache, half each for instructions and data. The relatively large cache (the IBM 3090 provides 128 to 256 Kbytes of cache) enables the system to keep large sets of program code and data local to the processor, off-loading the main memory bus and avoiding the need for a large register file with the accompanying windowing logic.

Instruction Set

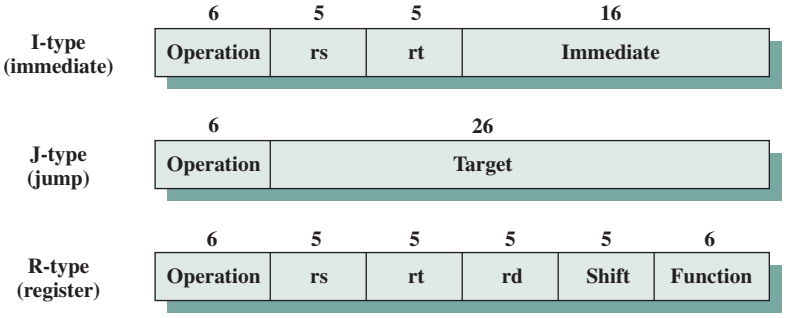
All MIPS R series instructions are encoded in a single 32-bit word format. All data operations are register to register; the only memory references are pure load/store operations.

The R4000 makes no use of condition codes. If an instruction generates a condition, the corresponding flags are stored in a general-purpose register. This avoids the need for special logic to deal with condition codes, as they affect the pipelining mechanism and the reordering of instructions by the compiler. Instead, the mechanisms already implemented to deal with register-value dependencies are employed. Further, conditions mapped onto the register files are subject to the same compile-time optimizations in allocation and reuse as other values stored in registers.

As with most RISC-based machines, the MIPS uses a single 32-bit instruction length. This single instruction length simplifies instruction fetch and decode, and it also simplifies the interaction of instruction fetch with the virtual memory management unit (i.e., instructions do not cross word or page boundaries). The three instruction formats (Figure 15.9) share common formatting of opcodes and register references, simplifying instruction decode. The effect of more complex instructions can be synthesized at compile time.

Only the simplest and most frequently used memory-addressing mode is implemented in hardware. All memory references consist of a 16-bit offset from a 32-bit register. For example, the "load word" instruction is of the form

```
lw r2, 128(r3)   /* load word at address 128 offset from
                    register 3 into register 2
```



Operation	**Operation code**
rs	**Source register specifier**
rt	**Source/destination register specifier**
Immediate	**Immediate, branch, or address displacement**
Target	**Jump target address**
rd	**Destination register specifier**
Shift	**Shift amount**
Function	**ALU/shift function specifier**

Figure 15.9 MIPS Instruction Formats

Each of the 32 general-purpose registers can be used as the base register. One register, r0, always contains 0.

The compiler makes use of multiple machine instructions to synthesize typical addressing modes in conventional machines. Here is an example from [CHOW87], which uses the instruction lui (load upper immediate). This instruction loads the upper half of a register with a 16-bit immediate value, setting the lower half to zero. Consider an assembly-language instruction that uses a 32-bit immediate argument

```
lw r2, #imm(r4)   /* load word at address using a 32-bit
                     immediate offset #imm
                     /* offset from register 4 into register 2
```

This instruction can be compiled into the following MIPS instructions

```
lui r1, #imm-hi      /* where #imm-hi is the high-order
                        16 bits of #imm
addu r1, r1, r4      /* add unsigned #imm-hi to r4 and
                        put in r1
lw r2, #imm-lo(r1)   /* where #imm-lo is the low-order
                        16 bits of #imm
```

Instruction Pipeline

With its simplified instruction architecture, the MIPS can achieve very efficient pipelining. It is instructive to look at the evolution of the MIPS pipeline, as it illustrates the evolution of RISC pipelining in general.

The initial experimental RISC systems and the first generation of commercial RISC processors achieve execution speeds that approach one instruction per system clock cycle. To improve on this performance, two classes of processors have evolved

to offer execution of multiple instructions per clock cycle: superscalar and superpipelined architectures. In essence, a superscalar architecture replicates each of the pipeline stages so that two or more instructions at the same stage of the pipeline can be processed simultaneously. A superpipelined architecture is one that makes use of more, and more fine-grained, pipeline stages. With more stages, more instructions can be in the pipeline at the same time, increasing parallelism.

Both approaches have limitations. With superscalar pipelining, dependencies between instructions in different pipelines can slow down the system. Also, overhead logic is required to coordinate these dependencies. With superpipelining, there is overhead associated with transferring instructions from one stage to the next.

Chapter 16 is devoted to a study of superscalar architecture. The MIPS R4000 is a good example of a RISC-based superpipeline architecture.

MIPS R3000 Five-Stage Pipeline Simulator

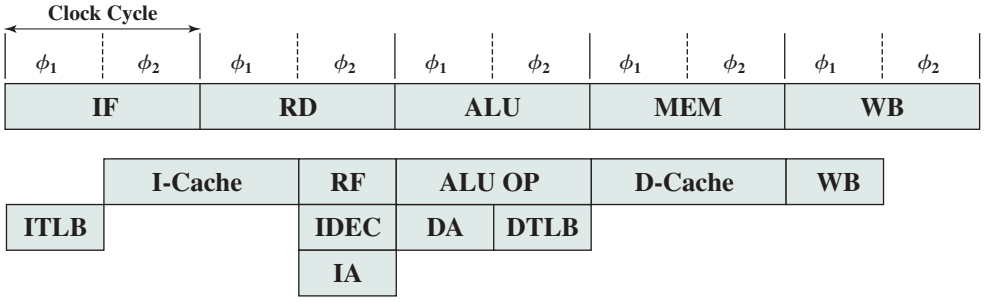
Figure 15.10a shows the instruction pipeline of the R3000. In the R3000, the pipeline advances once per clock cycle. The MIPS compiler is able to reorder instructions to fill delay slots with code 70 to 90% of the time. All instructions follow the same sequence of five pipeline stages:

- Instruction fetch;
- Source operand fetch from register file;
- ALU operation or data operand address generation;
- Data memory reference;
- Write back into register file.

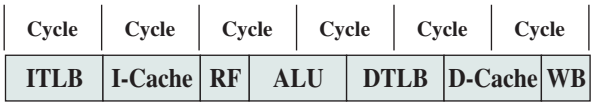
As illustrated in Figure 15.10a, there is not only parallelism due to pipelining but also parallelism within the execution of a single instruction. The 60-ns clock cycle is divided into two 30-ns stages. The external instruction and data access operations to the cache each require 60 ns, as do the major internal operations (OP, DA, IA). Instruction decode is a simpler operation, requiring only a single 30-ns stage, overlapped with register fetch in the same instruction. Calculation of an address for a branch instruction also overlaps instruction decode and register fetch, so that a branch at instruction i can address the ICACHE access of instruction $i + 2$. Similarly, a load at instruction i fetches data that are immediately used by the OP of instruction $i + 1$, while an ALU/shift result gets passed directly into instruction $i + 1$ with no delay. This tight coupling between instructions makes for a highly efficient pipeline.

In detail, then, each clock cycle is divided into separate stages, denoted as $\phi 1$ and $\phi 2$. The functions performed in each stage are summarized in Table 15.9.

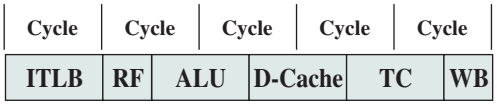
The R4000 incorporates a number of technical advances over the R3000. The use of more advanced technology allows the clock cycle time to be cut in half, to



(a) Detailed R3000 pipeline



(b) Modified R3000 pipeline with reduced latencies



(c) Optimized R3000 pipeline with parallel TLB and cache accesses

IF = **Instruction fetch**
RD = **Read**
MEM = **Memory access**
WB = **Write back to register file**
I-Cache = **Instruction cache access**
RF = **Fetch operand from register**
D-Cache = **Data cache access**
ITLB = **Instruction address translation**
IDEC = **Instruction decode**
IA = **Compute instruction address**
DA = **Calculate data virtual address**
DTLB = **Data address translation**
TC = **Data cache tag check**

Figure 15.10 Enhancing the R3000 Pipeline

Table 15.9 R3000 Pipeline Stages

Pipeline Stage	Phase	Function
IF	$\phi 1$	Using the TLB, translate an instruction virtual address to a physical address (after a branching decision).
IF	$\phi 2$	Send the physical address to the instruction address.
RD	$\phi 1$	Return instruction from instruction cache.
		Compare tags and validity of fetched instruction.
RD	$\phi 2$	Decode instruction.
		Read register file.
		If branch, calculate branch target address.
ALU	$\phi 1 + \phi 2$	If register-to-register operation, the arithmetic or logical operation is performed.
ALU	$\phi 1$	If a branch, decide whether the branch is to be taken or not.
		If a memory reference (load or store), calculate data virtual address.
ALU	$\phi 2$	If a memory reference, translate data virtual address to physical using TLB.
MEM	$\phi 1$	If a memory reference, send physical address to data cache.
MEM	$\phi 2$	If a memory reference, return data from data cache, and check tags.
WB	$\phi 1$	Write to register file.

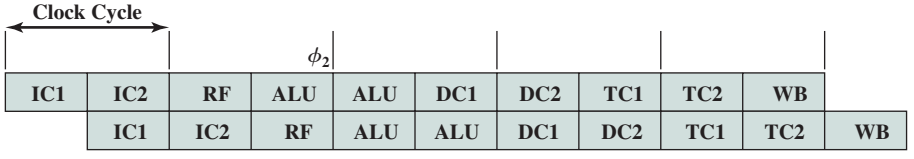
30 ns, and for the access time to the register file to be cut in half. In addition, there is greater density on the chip, which enables the instruction and data caches to be incorporated on the chip. Before looking at the final R4000 pipeline, let us consider how the R3000 pipeline can be modified to improve performance using R4000 technology.

Figure 15.10b shows a first step. Remember that the cycles in this figure are half as long as those in Figure 15.10a. Because they are on the same chip, the instruction and data cache stages take only half as long; so they still occupy only one clock cycle. Again, because of the speedup of the register file access, register read and write still occupy only half of a clock cycle.

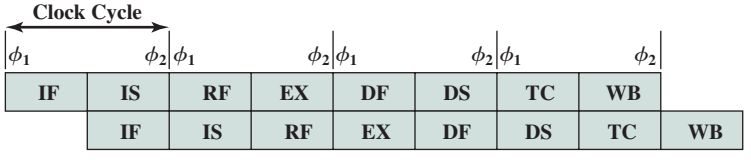
Because the R4000 caches are on-chip, the virtual-to-physical address translation can delay the cache access. This delay is reduced by implementing virtually indexed caches and going to a parallel cache access and address translation. Figure 15.10c shows the optimized R3000 pipeline with this improvement. Because of the compression of events, the data cache tag check is performed separately on the next cycle after cache access. This check determines whether the data item is in the cache.

In a superpipelined system, existing hardware is used several times per cycle by inserting pipeline registers to split up each pipe stage. Essentially, each superpipeline stage operates at a multiple of the base clock frequency, the multiple depending on the degree of superpipelining. The R4000 technology has the speed and density to permit superpipelining of degree 2. Figure 15.11a shows the optimized R3000 pipeline using this superpipelining. Note that this is essentially the same dynamic structure as Figure 15.10c.

Further improvements can be made. For the R4000, a much larger and specialized adder was designed. This makes it possible to execute ALU operations at



(a) Superpipelined implementation of the optimized R3000 pipeline



(b) R4000 pipeline

IF = Instruction fetch first half
IS = Instruction fetch second half
RF = Fetch operands from register
EX = Instruction execute
IC = Instruction cache
DC = Data cache
DF = Data cache first half
DS = Data cache second half
TC = Tag check
WB = Write back to register file

Figure 15.11 Theoretical R3000 and Actual R4000 Superpipelines

twice the rate. Other improvements allow the execution of loads and stores at twice the rate. The resulting pipeline is shown in Figure 15.11b.

The R4000 has eight pipeline stages, meaning that as many as eight instructions can be in the pipeline at the same time. The pipeline advances at the rate of two stages per clock cycle. The eight pipeline stages are as follows:

- **Instruction fetch first half:** Virtual address is presented to the instruction cache and the translation lookaside buffer.
- **Instruction fetch second half:** Instruction cache outputs the instruction and the TLB generates the physical address.
- **Register file:** Three activities occur in parallel:
  - —Instruction is decoded and check made for interlock conditions (i.e., this instruction depends on the result of a preceding instruction).
  - —Instruction cache tag check is made.
  - —Operands are fetched from the register file.
- **Instruction execute:** One of three activities can occur:
  - —If the instruction is a register-to-register operation, the ALU performs the arithmetic or logical operation.
  - —If the instruction is a load or store, the data virtual address is calculated.
  - —If the instruction is a branch, the branch target virtual address is calculated and branch conditions are checked.
- **Data cache first:** Virtual address is presented to the data cache and TLB.
- **Data cache second:** The TLB generates the physical address, and the data cache outputs the data.
- **Tag check:** Cache tag checks are performed for loads and stores.
- **Write back:** Instruction result is written back to register file.

15.7 SPARC

SPARC (Scalable Processor Architecture) refers to an architecture defined by Sun Microsystems. Sun developed its own SPARC implementation but also licenses the architecture to other vendors to produce SPARC-compatible machines. The SPARC architecture is inspired by the Berkeley RISC I machine, and its instruction set and register organization is based closely on the Berkeley RISC model.

SPARC Register Set

As with the Berkeley RISC, the SPARC makes use of register windows. Each window gives addressability to 24 registers, and the total number of windows is implementation dependent and ranges from 2 to 32 windows. Figure 15.12 illustrates an implementation that supports 8 windows, using a total of 136 physical registers; as the discussion in Section 15.2 indicates, this seems a reasonable number of windows. Physical registers 0 through 7 are global registers shared by all procedures.

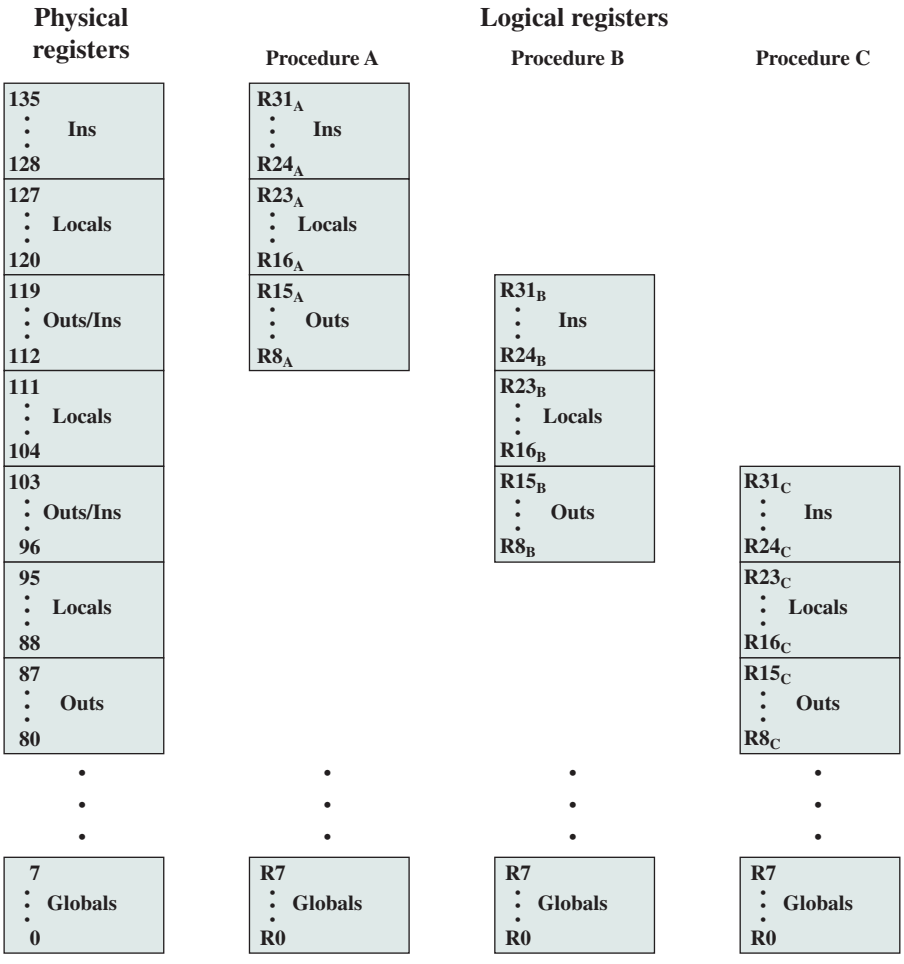


Figure 15.12 SPARC Register Window Layout with Three Procedures

Each procedure sees logical registers 0 through 31. Logical registers 24 through 31, referred to as *ins,* are shared with the calling (parent) procedure; and logical registers 8 through 15, referred to as *outs*, are shared with any called (child) procedure. These two portions overlap with other windows. Logical registers 16 through 23, referred to as *locals,* are not shared and do not overlap with other windows. Again, as the discussion of Section 12.1 indicates, the availability of 8 registers for parameter passing should be adequate in most cases (e.g., see Table 15.4).

Figure 15.13 is another view of the register overlap. The calling procedure places any parameters to be passed in its *outs* registers; the called procedure treats these same physical registers as it *ins* registers. The processor maintains a current window pointer (CWP), located in the processor status register (PSR), that points to the window of the currently executing procedure. The window invalid mask (WIM), also in the PSR, indicates which windows are invalid.

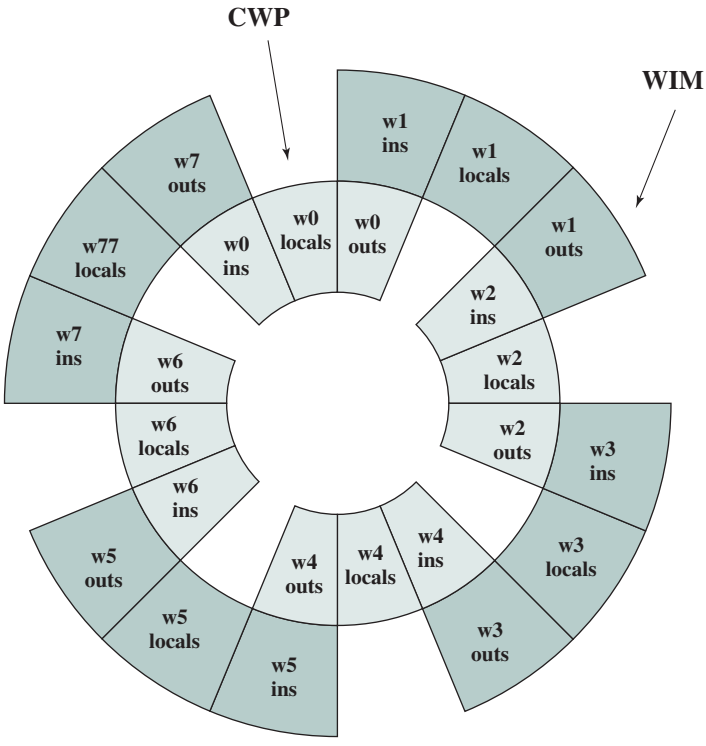


Figure 15.13 Eight Register Windows Forming a Circular Stack in SPARC

With the SPARC register architecture, it is usually not necessary to save and restore registers for a procedure call. The compiler is simplified because the compiler need be concerned only with allocating the local registers for a procedure in an efficient manner and need not be concerned with register allocation between procedures.

Instruction Set

Most of the SPARC instructions reference only register operands. Register-to-register instructions have three operands and can be expressed in the form

$$R_d \rightarrow R_{S1} \text{ op } S2$$

where R_d and R_{S1} are register references; S2 can refer either to a register or to a 13-bit immediate operand. Register zero (R_0) is hardwired with the value 0. This form is well suited to typical programs, which have a high proportion of local scalars and constants.

The available ALU operations can be grouped as follows:

- Integer addition (with or without carry).
- Integer subtraction (with or without carry).
- Bitwise Boolean AND, OR, XOR and their negations.
- Shift left logical, right logical, or right arithmetic.

All of these instructions, except the shifts, can optionally set the four condition codes (ZERO, NEGATIVE, OVERFLOW, CARRY). Signed integers are represented in 32-bit twos complement form.

Only simple load and store instructions reference memory. There are separate load and store instructions for word (32 bits), doubleword, halfword, and byte. For the latter two cases, there are instructions for loading these quantities as signed or unsigned numbers. Signed numbers are sign extended to fill out the 32-bit destination register. Unsigned numbers are padded with zeros.

The only available addressing mode, other than register, is a displacement mode. That is, the effective address (EA) of an operand consists of a displacement from an address contained in a register:

$$\begin{aligned} EA &= (R_{S1}) + S2 \\ \text{or } EA &= (R_{S1}) + (R_{S2}) \end{aligned}$$

depending on whether the second operand is immediate or a register reference. To perform a load or store, an extra stage is added to the instruction cycle. During the second stage, the memory address is calculated using the ALU; the load or store occurs in a third stage. This single addressing mode is quite versatile and can be used to synthesize other addressing modes, as indicated in Table 15.10.

It is instructive to compare the SPARC addressing capability with that of the MIPS. The MIPS makes use of a 16-bit offset, compared with a 13-bit offset on the SPARC. On the other hand, the MIPS does not permit an address to be constructed from the contents of two registers.

Instruction Format

As with the MIPS R4000, SPARC uses a simple set of 32-bit instruction formats (Figure 15.14). All instructions begin with a 2-bit opcode. For most instructions, this is extended with additional opcode bits elsewhere in the format. For the Call instruction, a 30-bit immediate operand is extended with two zero bits to the right to form a 32-bit PC-relative address in twos complement form. Instructions are aligned on a 32-bit boundary so that this form of addressing suffices.

The Branch instruction includes a 4-bit condition field that corresponds to the four standard condition code bits, so that any combination of conditions can be tested. The 22-bit PC-relative address is extended with two zero bits on the right to

Table 15.10 Synthesizing Other Addressing Modes with SPARC Addressing Modes

Instruction Type	Addressing Mode	Algorithm	SPARC Equivalent
Register-to-register	Immediate	operand = A	S2
Load, store	Direct	EA = A	$R_0 + S_2$
Register-to-register	Register	EA = R	R_{S1}, S_{S2}
Load, store	Register Indirect	EA = (R)	$R_{S1} + 0$
Load, store	Displacement	EA = (R) + A	$R_{S1} + S2$

Note: S2 = either a register operand or a 13-bit immediate operand.

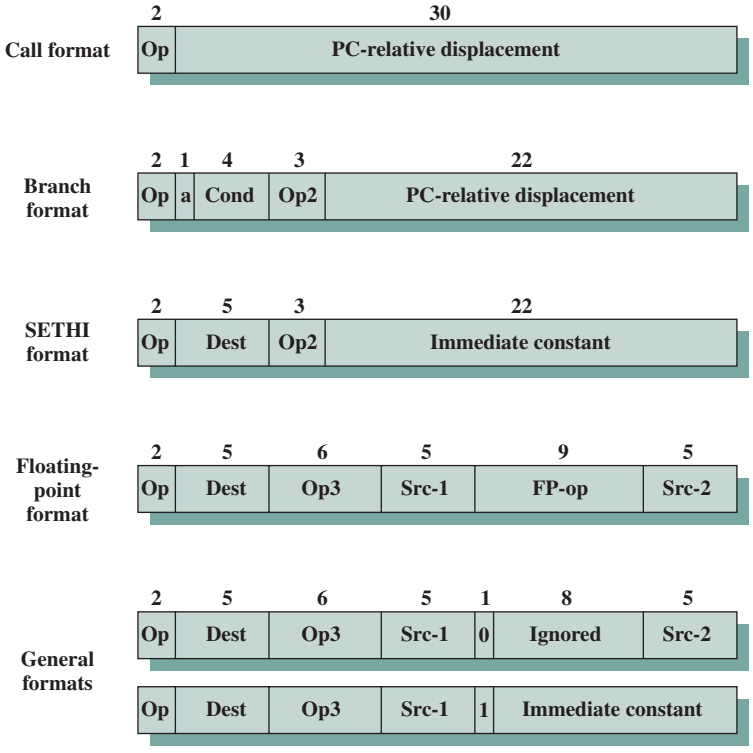


Figure 15.14 SPARC Instruction Formats

form a 24-bit twos complement relative address. An unusual feature of the Branch instruction is the annul bit. When the annul bit is not set, the instruction after the branch is always executed, regardless of whether the branch is taken. This is the typical delayed branch operation found on many RISC machines and described in Section 15.5 (see Figure 15.7). However, when the annul bit is set, the instruction following the branch is executed only if the branch is taken. The processor suppresses the effect of that instruction even though it is already in the pipeline. This annul bit is useful because it makes it easier for the compiler to fill the delay slot following a conditional branch. The instruction that is the target of the branch can always be put in the delay slot, because if the branch is not taken, the instruction can be annulled. The reason this technique is desirable is that conditional branches are generally taken more than half the time.

The SETHI instruction is a special instruction used to form a 32-bit constant. This feature is needed to form large data constants; for example, it can be used to form a large offset for a load or store instruction. The SETHI instruction sets the 22 high-order bits of a register with its 22-bit immediate operand, and zeros out the low-order 10 bits. An immediate constant of up to 13 bits can be specified in one of the general formats, and such an instruction could be used to fill in the remaining 10 bits of the register. A load or store instruction can also be used to achieve a direct

addressing mode. To load a value from location K in memory, we could use the following SPARC instructions:

```
sethi   %hi(K), %r8          ;load high-order 22 bits of address of location
                             ;K into register r8
Ld      [%r8 + %lo(K)], %r8  ;load contents of location K into r8
```

The macros %hi and %lo are used to define immediate operands consisting of the appropriate address bits of a location. This use of SETHI is similar to the use of the lui instruction on the MIPS.

The floating-point format is used for floating-point operations. Two source and one destination registers are designated.

Finally, all other operations, including loads, stores, arithmetic, and logical operations use one of the last two formats shown in Figure 15.14. One of the formats makes use of two source registers and a destination register, while the other uses one source register, one 13-bit immediate operand, and one destination register.

15.8 RISC VERSUS CISC CONTROVERSY

For many years, the general trend in computer architecture and organization has been toward increasing processor complexity: more instructions, more addressing modes, more specialized registers, and so on. The RISC movement represents a fundamental break with the philosophy behind that trend. Naturally, the appearance of RISC systems, and the publication of papers by its proponents extolling RISC virtues, led to a reaction from those involved in the design of CISC architectures.

The work that has been done on assessing merits of the RISC approach can be grouped into two categories:

- **Quantitative:** Attempts to compare program size and execution speed of programs on RISC and CISC machines that use comparable technology.
- **Qualitative:** Examines issues such as high-level language support and optimum use of VLSI real estate.

Most of the work on quantitative assessment has been done by those working on RISC systems [PATT82b, HEAT84, PATT84], and it has been, by and large, favorable to the RISC approach. Others have examined the issue and come away unconvinced [COLW85a, FLYN87, DAVI87]. There are several problems with attempting such comparisons [SERL86]:

- There is no pair of RISC and CISC machines that are comparable in life-cycle cost, level of technology, gate complexity, sophistication of compiler, operating system support, and so on.
- No definitive test set of programs exists. Performance varies with the program.
- It is difficult to sort out hardware effects from effects due to skill in compiler writing.
- Most of the comparative analysis on RISC has been done on "toy" machines rather than commercial products. Furthermore, most commercially available

machines advertised as RISC possess a mixture of RISC and CISC characteristics. Thus, a fair comparison with a commercial, "pure-play" CISC machine (e.g., VAX, Pentium) is difficult.

The qualitative assessment is, almost by definition, subjective. Several researchers have turned their attention to such an assessment [COLW85a, WALL85], but the results are, at best, ambiguous, and certainly subject to rebuttal [PATT85b] and, of course, counterrebuttal [COLW85b].

In more recent years, the RISC versus CISC controversy has died down to a great extent. This is because there has been a gradual convergence of the technologies. As chip densities and raw hardware speeds increase, RISC systems have become more complex. At the same time, in an effort to squeeze out maximum performance, CISC designs have focused on issues traditionally associated with RISC, such as an increased number of general-purpose registers and increased emphasis on instruction pipeline design.

15.9 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

complex instruction set computer (CISC)	high-level language (HLL)	register window
delayed branch	reduced instruction set computer (RISC)	SPARC
delayed load	register file	

Review Questions

15.1 What are some typical distinguishing characteristics of RISC organization?

15.2 Briefly explain the two basic approaches used to minimize register-memory operations on RISC machines.

15.3 If a circular register buffer is used to handle local variables for nested procedures, describe two approaches for handling global variables.

15.4 What are some typical characteristics of a RISC instruction set architecture?

15.5 What is a delayed branch?

Problems

15.1 Considering the call-return pattern in Figure 4.21, how many overflows and underflows (each of which causes a register save/restore) will occur with a window size of
- **a.** 5?
- **b.** 8?
- **c.** 16?

15.2 In the discussion of Figure 15.2, it was stated that only the first two portions of a window are saved or restored. Why is it not necessary to save the temporary registers?

15.3 We wish to determine the execution time for a given program using the various pipelining schemes discussed in Section 15.5. Let

N = number of executed instructions
D = number of memory accesses
J = number of jump instructions

For the simple sequential scheme (Figure 15.6a), the execution time is $2N + D$ stages. Derive formulas for two-stage, three-stage, and four-stage pipelining.

15.4 Reorganize the code sequence in Figure 15.6d to reduce the number of NOOPs.

15.5 Consider the following code fragment in a high-level language:

```
for I in 1…100 loop
    S ← S + Q(I). VAL
end loop;
```

Assume that Q is an array of 32-byte records and the VAL field is in the first 4 bytes of each record. Using x86 code, we can compile this program fragment as follows:

```
      MOV    ECX,1          ;use register ECX to hold I
LP:   IMUL   EAX, ECX, 32   ;get offset in EAX
      MOV    EBX, Q[EAX]    ;load VAL field
      ADD    S, EBX         ;add to S
      INC    ECX            ;increment I
      CMP    ECX, 101       :compare to 101
      JNE    LP             ;loop until I = 100
```

This program makes use of the IMUL instruction, which multiplies the second operand by the immediate value in the third operand and places the result in the first operand (see Problem 10.13). A RISC advocate would like to demonstrate that a clever compiler can eliminate unnecessarily complex instructions such as IMUL. Provide the demonstration by rewriting the above x86 program without using the IMUL instruction.

15.6 Consider the following loop:

```
S := 0;
for K:=1 to 100 do
      S:=S − K;
```

A straightforward translation of this into a generic assembly language would look something like this:

```
      LD     R1, 0          ;keep value of S in R1
      LD     R2,1           ;keep value of K in R2
LP    SUB    R1, R1, R2     ;S:=S − K
      BEQ    R2, 100, EXIT  ;done if K = 100
      ADD    R2, R2, 1      ;else increment K
      JMP    LP             ;back to start of loop
```

A compiler for a RISC machine will introduce delay slots into this code so that the processor can employ the delayed branch mechanism. The JMP instruction is easy to deal with, because this instruction is always followed by the SUB instruction; therefore,

we can simply place a copy of the SUB instruction in the delay slot after the JMP. The BEQ presents a difficulty. We can't leave the code as is, because the ADD instruction would then be executed one too many times. Therefore, a NOP instruction is needed. Show the resulting code.

15.7 A RISC machine's compiler may do both a mapping of symbolic registers to actual registers and a rearrangement of instructions for pipeline efficiency. An interesting question arises as to the order in which these two operations should be done. Consider the following program fragment:

```
LD      SR1, A          ;load A into symbolic register 1
LD      SR2, B          ;load B into symbolic register 2
ADD     SR3, SR1, SR2   ;add contents of SR1 and SR2 and store in SR3
LD      SR4, C
LD      SR5, D
ADD     SR6, SR4, SR5
```

a. First do the register mapping and then any possible instruction reordering. How many machine registers are used? Has there been any pipeline improvement?
b. Starting with the original program, now do instruction reordering and then any possible mapping. How many machine registers are used? Has there been any pipeline improvement?

15.8 Add entries for the following processors to Table 15.7:
a. Pentium II
b. ARM

15.9 In many cases, common machine instructions that are not listed as part of the MIPS instruction set can be synthesized with a single MIPS instruction. Show this for the following:
a. Register-to-register move
b. Increment, decrement
c. Complement
d. Negate
e. Clear

15.10 A SPARC implementation has K register windows. What is the number N of physical registers?

15.11 SPARC is lacking a number of instructions commonly found on CISC machines. Some of these are easily simulated using either register R0, which is always set to 0, or a constant operand. These simulated instructions are called pseudoinstructions and are recognized by the SPARC assembler. Show how to simulate the following pseudoinstructions, each with a single SPARC instruction. In all of these, src and dst refer to registers. (*Hint*: A store to R0 has no effect.)

a. MOV src, dst
b. COMPARE src1, src2
c. TEST src1
d. NOT dst
e. NEG dst
f. INC dst
g. DEC dst
h. CLR dst
i. NOP

15.12 Consider the following code fragment:

```
if K > 10
    L := K + 1
else
    L := K – 1
```

A straightforward translation of this statement into SPARC assembler could take the following form:

```
      sethi   %hi(K), %r8             ;load high-order 22 bits of address of location
                                      ;K into register r8
      ld      [%r8 + %lo(K)], %r8     ;load contents of location K into r8
      cmp     %r8, 10                 ;compare contents of r8 with 10
      ble     L1                      ;branch if (r8) ≤ 10
      nop
      sethi   %hi(K), %r9
      ld      [%r9 + %lo(K)], %r9     ;load contents of location K into r9
      inc     %r9                     ;add 1 to (r9)
      sethi   %hi(L), %r10
      st      %r9, [%r10 + %lo(L)]    ;store (r9) into location L
      b       L2
      nop
L1:   sethi   %hi(K), %r11
      ld      [%r11 + %lo(K)], %r12   ;load contents of location K into r12
      dec     %r12                    ;subtract 1 from (r12)
      sethi   %hi(L), %r13
      st      %r12, [%r13 + %lo(L)]   ;store (r12) into location L
L2:
```

The code contains a nop after each branch instruction to permit delayed branch operation.

a. Standard compiler optimizations that have nothing to do with RISC machines are generally effective in being able to perform two transformations on the foregoing code. Notice that two of the loads are unnecessary and that the two stores can be merged if the store is moved to a different place in the code. Show the program after making these two changes.

b. It is now possible to perform some optimizations peculiar to SPARC. The nop after the ble can be replaced by moving another instruction into that delay slot and setting the annul bit on the ble instruction (expressed as ble,a L1). Show the program after this change.

c. There are now two unnecessary instructions. Remove these and show the resulting program.

CHAPTER 16

INSTRUCTION-LEVEL PARALLELISM AND SUPERSCALAR PROCESSORS

16.1 Overview
- Superscalar versus Superpipelined
- Constraints

16.2 Design Issues
- Instruction-Level Parallelism and Machine Parallelism
- Instruction Issue Policy
- Register Renaming
- Machine Parallelism
- Branch Prediction
- Superscalar Execution
- Superscalar Implementation

16.3 Intel Core Microarchitecture
- Front End
- Out-of-Order Execution Logic
- Integer and Floating-Point Execution Units

16.4 Arm Cortex-A8
- Instruction Fetch Unit
- Instruction Decode Unit
- Integer Execute Unit
- SIMD and Floating-Point Pipeline

16.5 ARM Cortex-M3
- Pipeline Structure
- Dealing with Branches

16.6 Key Terms, Review Questions, and Problems

Learning Objectives

After studying this chapter, you should be able to:

- Explain the difference between superscalar and superpipelined approaches.
- Define instruction-level parallelism.
- Discuss dependencies and resource conflicts as limitations to instruction-level parallelism
- Present an overview of the design issues involved in instruction-level parallelism.
- Compare and contrast techniques of improving pipeline performance in RISC machines and superscalar machines.

A superscalar implementation of a processor architecture is one in which common instructions—integer and floating-point arithmetic, loads, stores, and conditional branches—can be initiated simultaneously and executed independently. Such implementations raise a number of complex design issues related to the instruction pipeline.

Superscalar design arrived on the scene hard on the heels of RISC architecture. Although the simplified instruction set architecture of a RISC machine lends itself readily to superscalar techniques, the superscalar approach can be used on either a RISC or CISC architecture.

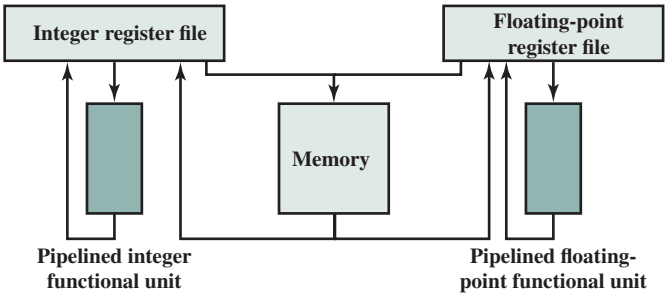
Whereas the gestation period for the arrival of commercial RISC machines from the beginning of true RISC research with the IBM 801 and the Berkeley RISC I was seven or eight years, the first superscalar machines became commercially available within just a year or two of the coining of the term **superscalar**. The superscalar approach has now become the standard method for implementing high-performance microprocessors.

In this chapter, we begin with an overview of the superscalar approach, contrasting it with superpipelining. Next, we present the key design issues associated with superscalar implementation. Then we look at several important examples of superscalar architecture.

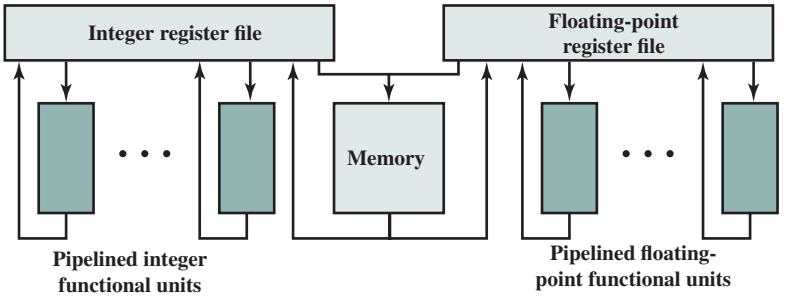
16.1 OVERVIEW

The term *superscalar*, first coined in 1987 [AGER87], refers to a machine that is designed to improve the performance of the execution of scalar instructions. In most applications, the bulk of the operations are on scalar quantities. Accordingly, the superscalar approach represents the next step in the evolution of high-performance general-purpose processors.

The essence of the superscalar approach is the ability to execute instructions independently and concurrently in different pipelines. The concept can be further exploited by allowing instructions to be executed in an order different from the program order. Figure 16.1 compares, in general terms, the scalar and superscalar approaches. In a traditional scalar organization, there is a single pipelined functional unit for integer operations and one for floating-point operations. Parallelism is achieved by enabling multiple instructions to be at different stages of the pipeline at



(a) Scalar organization



(b) Superscalar organization

Figure 16.1 Superscalar Organization Compared to Ordinary Scalar Organization

one time. In the superscalar organization, there are multiple functional units, each of which is implemented as a pipeline. Each individual functional unit provides a degree of parallelism by virtue of its pipelined structure. The use of multiple functional units enables the processor to execute streams of instructions in parallel, one stream for each pipeline. It is the responsibility of the hardware, in conjunction with the compiler, to assure that the parallel execution does not violate the intent of the program.

Many researchers have investigated superscalar-like processors, and their research indicates that some degree of performance improvement is possible. Table 16.1 presents the reported performance advantages. The differences in the

Table 16.1 Reported Speedups of Superscalar-Like Machines

Reference	Speedup
[TJAD70]	1.8
[KUCK77]	8
[WEIS84]	1.58
[ACOS86]	2.7
[SOHI90]	1.8
[SMIT89]	2.3
[JOUP89b]	2.2
[LEE91]	7

results arise from differences both in the hardware of the simulated machine and in the applications being simulated.

Superscalar versus Superpipelined

An alternative approach to achieving greater performance is referred to as superpipelining, a term first coined in 1988 [JOUP88]. Superpipelining exploits the fact that many pipeline stages perform tasks that require less than half a clock cycle. Thus, a doubled internal clock speed allows the performance of two tasks in one external clock cycle. We have seen one example of this approach with the MIPS R4000.

Figure 16.2 compares the two approaches. The upper part of the diagram illustrates an ordinary pipeline, used as a base for comparison. The base pipeline issues

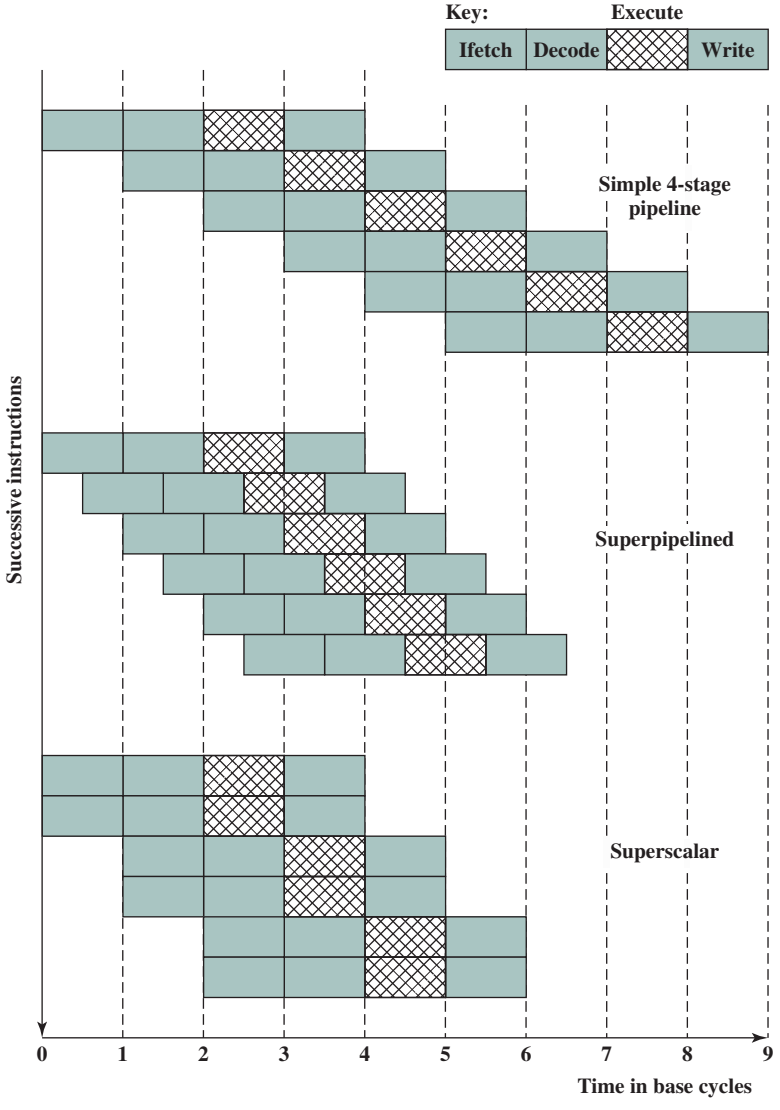


Figure 16.2 Comparison of Superscalar and Superpipeline Approaches

one instruction per clock cycle and can perform one pipeline stage per clock cycle. The pipeline has four stages: instruction fetch; operation decode; operation execution; and result write back. The execution stage is crosshatched for clarity. Note that although several instructions are executing concurrently, only one instruction is in its execution stage at any one time.

The next part of the diagram shows a **superpipelined** implementation that is capable of performing two pipeline stages per clock cycle. An alternative way of looking at this is that the functions performed in each stage can be split into two nonoverlapping parts and each can execute in half a clock cycle. A superpipeline implementation that behaves in this fashion is said to be of degree 2. Finally, the lowest part of the diagram shows a superscalar implementation capable of executing two instances of each stage in parallel. Higher-degree superpipeline and superscalar implementations are of course possible.

Both the superpipeline and the superscalar implementations depicted in Figure 16.2 have the same number of instructions executing at the same time in the steady state. The superpipelined processor falls behind the superscalar processor at the start of the program and at each branch target.

Constraints

The superscalar approach depends on the ability to execute multiple instructions in parallel. The term **instruction-level parallelism** refers to the degree to which, on average, the instructions of a program can be executed in parallel. A combination of compiler-based optimization and hardware techniques can be used to maximize instruction-level parallelism. Before examining the design techniques used in superscalar machines to increase instruction-level parallelism, we need to look at the fundamental limitations to parallelism with which the system must cope. [JOHN91] lists five limitations:

- True data dependency;
- Procedural dependency;
- Resource conflicts;
- Output dependency;
- Antidependency.

We examine the first three of these limitations in the remainder of this section. A discussion of the last two must await some of the developments in the next section.

TRUE DATA DEPENDENCY Consider the following sequence:[1]

```
ADD EAX, ECX ;load register EAX with the con-
             ;tents of ECX plus the contents
             ;of EAX
MOV EBX, EAX ;load EBX with the contents of EAX
```

The second instruction can be fetched and decoded but cannot execute until the first instruction executes. The reason is that the second instruction needs data

[1]For the Intel x86 assembly language, a semicolon starts a comment field.

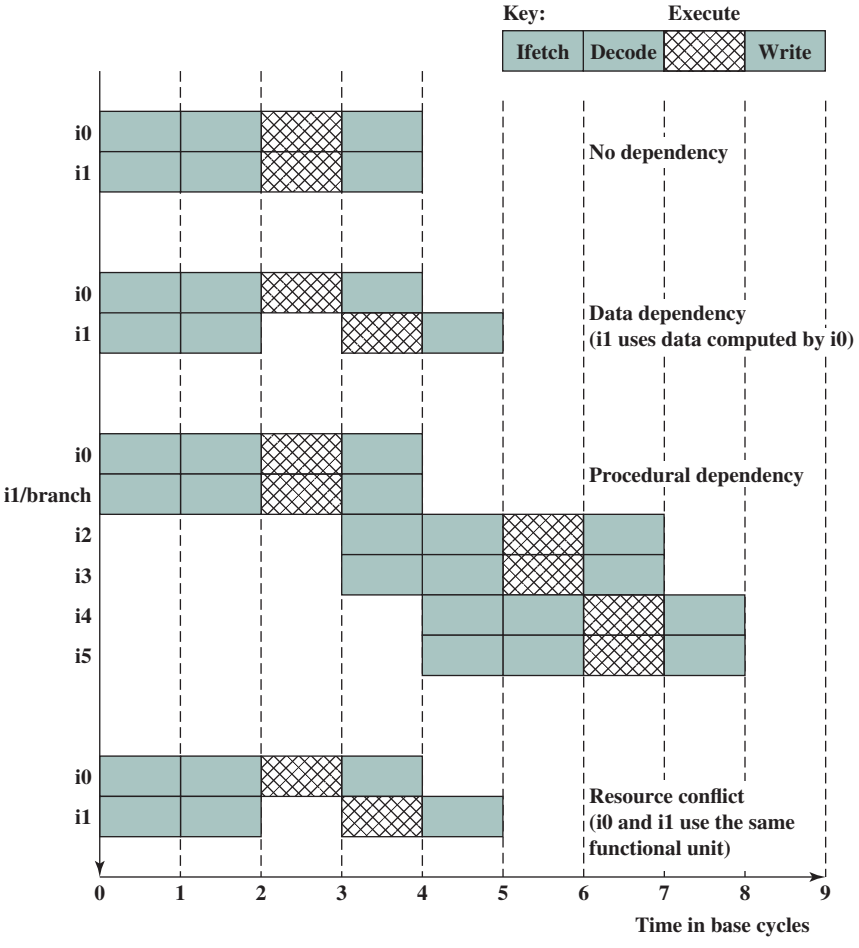


Figure 16.3 Effect of Dependencies

produced by the first instruction. This situation is referred to as a **true data dependency** (also called **flow dependency** or **read after write [RAW] dependency**).

Figure 16.3 illustrates this **dependency** in a superscalar machine of degree 2. With no dependency, two instructions can be fetched and executed in parallel. If there is a data dependency between the first and second instructions, then the second instruction is delayed as many clock cycles as required to remove the dependency. In general, any instruction must be delayed until all of its input values have been produced.

In a simple pipeline, such as illustrated in the upper part of Figure 16.2, the aforementioned sequence of instructions would cause no delay. However, consider the following, in which one of the loads is from memory rather than from a register:

```
MOV EAX, eff ;load register EAX with the
             contents of effective memory
             address eff
MOV EBX, EAX ;load EBX with the contents of EAX
```

A typical RISC processor takes two or more cycles to perform a load from memory when the load is a cache hit. It can take tens or even hundreds of cycles for a cache miss on all cache levels, because of the delay of an off-chip memory access. One way to compensate for this delay is for the compiler to reorder instructions so that one or more subsequent instructions that do not depend on the memory load can begin flowing through the pipeline. This scheme is less effective in the case of a superscalar pipeline: The independent instructions executed during the load are likely to be executed on the first cycle of the load, leaving the processor with nothing to do until the load completes.

PROCEDURAL DEPENDENCIES As was discussed in Chapter 14, the presence of branches in an instruction sequence complicates the pipeline operation. The instructions following a branch (taken or not taken) have a **procedural dependency** on the branch and cannot be executed until the branch is executed. Figure 16.3 illustrates the effect of a branch on a superscalar pipeline of degree 2.

As we have seen, this type of procedural dependency also affects a scalar pipeline. The consequence for a superscalar pipeline is more severe, because a greater magnitude of opportunity is lost with each delay.

If variable-length instructions are used, then another sort of procedural dependency arises. Because the length of any particular instruction is not known, it must be at least partially decoded before the following instruction can be fetched. This prevents the simultaneous fetching required in a superscalar pipeline. This is one of the reasons that superscalar techniques are more readily applicable to a RISC or RISC-like architecture, with its fixed instruction length.

RESOURCE CONFLICT A **resource conflict** is a competition of two or more instructions for the same resource at the same time. Examples of resources include memories, caches, buses, register-file ports, and functional units (e.g., ALU adder).

In terms of the pipeline, a resource conflict exhibits similar behavior to a data dependency (Figure 16.3). There are some differences, however. For one thing, resource conflicts can be overcome by duplication of resources, whereas a true data dependency cannot be eliminated. Also, when an operation takes a long time to complete, resource conflicts can be minimized by pipelining the appropriate functional unit.

16.2 DESIGN ISSUES

Instruction-Level Parallelism and Machine Parallelism

[JOUP89a] makes an important distinction between the two related concepts of instruction-level parallelism and machine parallelism. **Instruction-level parallelism** exists when instructions in a sequence are independent and thus can be executed in parallel by overlapping.

As an example of the concept of instruction-level parallelism, consider the following two code fragments [JOUP89b]:

```
Load R1 ← R2          Add R3 ← R3, "1"
Add R3 ← R3, "1"      Add R4 ← R3, R2
Add R4 ← R4, R2       Store [R4] ← R0
```

The three instructions on the left are independent, and in theory all three could be executed in parallel. In contrast, the three instructions on the right cannot be executed in parallel because the second instruction uses the result of the first, and the third instruction uses the result of the second.

The degree of instruction-level parallelism is determined by the frequency of true data dependencies and procedural dependencies in the code. These factors, in turn, are dependent on the instruction set architecture and on the application. Instruction-level parallelism is also determined by what [JOUP89a] refers to as operation latency: the time until the result of an instruction is available for use as an operand in a subsequent instruction. The latency determines how much of a delay a data or procedural dependency will cause.

Machine parallelism is a measure of the ability of the processor to take advantage of instruction-level parallelism. Machine parallelism is determined by the number of instructions that can be fetched and executed at the same time (the number of parallel pipelines) and by the speed and sophistication of the mechanisms that the processor uses to find independent instructions.

Both instruction-level and machine parallelism are important factors in enhancing performance. A program may not have enough instruction-level parallelism to take full advantage of machine parallelism. The use of a fixed-length instruction set architecture, as in a RISC, enhances instruction-level parallelism. On the other hand, limited machine parallelism will limit performance no matter what the nature of the program.

Instruction Issue Policy

As was mentioned, machine parallelism is not simply a matter of having multiple instances of each pipeline stage. The processor must also be able to identify instruction-level parallelism and orchestrate the fetching, decoding, and execution of instructions in parallel. [JOHN91] uses the term **instruction issue** to refer to the process of initiating instruction execution in the processor's functional units and the term **instruction issue policy** to refer to the protocol used to issue instructions. In general, we can say that instruction issue occurs when instruction moves from the decode stage of the pipeline to the first execute stage of the pipeline.

In essence, the processor is trying to look ahead of the current point of execution to locate instructions that can be brought into the pipeline and executed. Three types of orderings are important in this regard:

- The order in which instructions are fetched;
- The order in which instructions are executed;
- The order in which instructions update the contents of register and memory locations.

The more sophisticated the processor, the less it is bound by a strict relationship between these orderings. To optimize utilization of the various pipeline elements, the processor will need to alter one or more of these orderings with respect to the ordering to be found in a strict sequential execution. The one constraint on the processor is that the result must be correct. Thus, the processor must accommodate the various dependencies and conflicts discussed earlier.

In general terms, we can group superscalar instruction issue policies into the following categories:

- In-order issue with in-order completion.
- In-order issue with **out-of-order** completion.
- Out-of-order issue with out-of-order completion.

IN-ORDER ISSUE WITH IN-ORDER COMPLETION The simplest instruction issue policy is to issue instructions in the exact order that would be achieved by sequential execution (**in-order issue**) and to write results in that same order (**in-order completion**). Not even scalar pipelines follow such a simple-minded policy. However, it is useful to consider this policy as a baseline for comparing more sophisticated approaches.

Figure 16.4a gives an example of this policy. We assume a superscalar pipeline capable of fetching and decoding two instructions at a time, having three separate functional units (e.g., two integer arithmetic and one floating-point arithmetic), and having two instances of the write-back pipeline stage. The example assumes the following constraints on a six-instruction code fragment:

- I1 requires two cycles to execute.
- I3 and I4 conflict for the same functional unit.
- I5 depends on the value produced by I4.
- I5 and I6 conflict for a functional unit.

Instructions are fetched two at a time and passed to the decode unit. Because instructions are fetched in pairs, the next two instructions must wait until the pair of decode pipeline stages has cleared. To guarantee in-order **completion**, when there is a conflict for a functional unit or when a functional unit requires more than one cycle to generate a result, the issuing of instructions temporarily stalls.

In this example, the elapsed time from decoding the first instruction to writing the last results is eight cycles.

IN-ORDER ISSUE WITH OUT-OF-ORDER COMPLETION **Out-of-order completion** is used in scalar RISC processors to improve the performance of instructions that require multiple cycles. Figure 16.4b illustrates its use on a superscalar processor. Instruction I2 is allowed to run to completion prior to I1. This allows I3 to be completed earlier, with the net result of a savings of one cycle.

With out-of-order completion, any number of instructions may be in the execution stage at any one time, up to the maximum degree of machine parallelism across all functional units. Instruction issuing is stalled by a resource conflict, a data dependency, or a procedural dependency.

In addition to the aforementioned limitations, a new dependency, which we referred to earlier as an **output dependency** (also called **write after write [WAW] dependency**), arises. The following code fragment illustrates this dependency (*op* represents any operation):

```
I1: R3 ← R3 op R5
I2: R4 ← R3 + 1
I3: R3 ← R5 + 1
I4: R7 ← R3 op R4
```

Decode		Execute			Write		Cycle
I1	I2						1
I3	I4	I1	I2				2
I3	I4	I1					3
	I4			I3	I1	I2	4
I5	I6			I4			5
	I6		I5		I3	I4	6
			I6				7
					I5	I6	8

(a) In-order issue and in-order completion

Decode		Execute			Write		Cycl e
I1	I2						1
I3	I4	I1	I2				2
	I4	I1		I3	I2		3
I5	I6			I4	I1	I3	4
	I6		I5		I4		5
			I6		I5		6
					I6		7

(b) In-order issue and out-of-order completion

Decode		Window	Execute			Write		Cycle
I1	I2							1
I3	I4	*I1,I2*	I1	I2				2
I5	I6	*I3,I4*	I1		I3	I2		3
		I4,I5,I6		I6	I4	I1	I3	4
		I5		I5		I4	I6	5
						I5		6

(c) Out-of-order issue and out-of-order completion

Figure 16.4 Superscalar Instruction Issue and Completion Policies

Instruction I2 cannot execute before instruction I1, because it needs the result in register R3 produced in I1; this is an example of a true data dependency, as described in Section 16.1. Similarly, I4 must wait for I3, because it uses a result produced by I3. What about the relationship between I1 and I3? There is no data dependency here, as we have defined it. However, if I3 executes to completion prior to I1, then the wrong value of the contents of R3 will be fetched for the execution of I4. Consequently, I3 must complete after I1 to produce the correct output values. To ensure this, the issuing of the third instruction must be stalled if its result might later be overwritten by an older instruction that takes longer to complete.

Out-of-order completion requires more complex instruction issue logic than in-order completion. In addition, it is more difficult to deal with instruction interrupts and exceptions. When an interrupt occurs, instruction execution at the current

point is suspended, to be resumed later. The processor must assure that the resumption takes into account that, at the time of interruption, instructions ahead of the instruction that caused the interrupt may already have completed.

OUT-OF-ORDER ISSUE WITH OUT-OF-ORDER COMPLETION With in-order issue, the processor will only decode instructions up to the point of a dependency or conflict. No additional instructions are decoded until the conflict is resolved. As a result, the processor cannot look ahead of the point of conflict to subsequent instructions that may be independent of those already in the pipeline and that may be usefully introduced into the pipeline.

To allow **out-of-order issue**, it is necessary to decouple the decode and execute stages of the pipeline. This is done with a buffer referred to as an **instruction window**. With this organization, after a processor has finished decoding an instruction, it is placed in the instruction window. As long as this buffer is not full, the processor can continue to fetch and decode new instructions. When a functional unit becomes available in the execute stage, an instruction from the instruction window may be issued to the execute stage. Any instruction may be issued, provided that (1) it needs the particular functional unit that is available, and (2) no conflicts or dependencies block this instruction. Figure 16.5 suggests this organization.

The result of this organization is that the processor has a lookahead capability, allowing it to identify independent instructions that can be brought into the execute stage. Instructions are issued from the instruction window with little regard for their original program order. As before, the only constraint is that the program execution behaves correctly.

Figures 16.4c illustrates this policy. During each of the first three cycles, two instructions are fetched into the decode stage. During each cycle, subject to the constraint of the buffer size, two instructions move from the decode stage to the instruction window. In this example, it is possible to issue instruction I6 ahead of I5 (recall that I5 depends on I4, but I6 does not). Thus, one cycle is saved in both the execute and write-back stages, and the end-to-end savings, compared with Figure 16.4b, is one cycle.

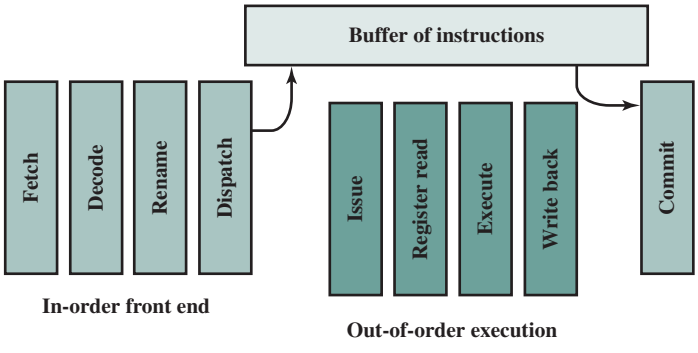


Figure 16.5 Organization for Out-of-Order Issue with Out-of-Order Completion

The instruction window is depicted in Figure 16.4c to illustrate its role. However, this window is not an additional pipeline stage. An instruction being in the window simply implies that the processor has sufficient information about that instruction to decide when it can be issued.

The out-of-order issue, out-of-order completion policy is subject to the same constraints described earlier. An instruction cannot be issued if it violates a dependency or conflict. The difference is that more instructions are available for issuing, reducing the probability that a pipeline stage will have to stall. In addition, a new dependency, which we referred to earlier as an **antidependency** (also called **write after read [WAR] dependency**), arises. The code fragment considered earlier illustrates this dependency:

```
I1: R3 ← R3 op R5
I2: R4 ← R3 + 1
I3: R3 ← R5 + 1
I4: R7 ← R3 op R4
```

Instruction I3 cannot complete execution before instruction I2 begins execution and has fetched its operands. This is so because I3 updates register R3, which is a source operand for I2. The term *antidependency* is used because the constraint is similar to that of a true data dependency, but reversed: Instead of the first instruction producing a value that the second instruction uses, the second instruction destroys a value that the first instruction uses.

Reorder Buffer Simulator
Tomasulo's Algorithm Simulator
Alternative Simulation of Tomasulo's Algorithm

One common technique that is used to support out-of-order completion is the reorder buffer. The reorder buffer is temporary storage for results completed out of order that are then committed to the register file in program order. A related concept is Tomasulo's algorithm. Appendix N examines these concepts.

Register Renaming

When out-of-order instruction issuing and/or out-of-order instruction completion are allowed, we have seen that this gives rise to the possibility of WAW dependencies and WAR dependencies. These dependencies differ from RAW data dependencies and resource conflicts, which reflect the flow of data through a program and the sequence of execution. WAW dependencies and WAR dependencies, on the other hand, arise because the values in registers may no longer reflect the sequence of values dictated by the program flow.

When instructions are issued in sequence and complete in sequence, it is possible to specify the contents of each register at each point in the execution. When out-of-order techniques are used, the values in registers cannot be fully known at each point in time just from a consideration of the sequence of instructions dictated

by the program. In effect, values are in conflict for the use of registers, and the processor must resolve those conflicts by occasionally stalling a pipeline stage.

Antidependencies and output dependencies are both examples of storage conflicts. Multiple instructions are competing for the use of the same register locations, generating pipeline constraints that retard performance. The problem is made more acute when register optimization techniques are used (as discussed in Chapter 15), because these compiler techniques attempt to maximize the use of registers, hence maximizing the number of storage conflicts.

One method for coping with these types of storage conflicts is based on a traditional resource-conflict solution: duplication of resources. In this context, the technique is referred to as **register renaming**. In essence, registers are allocated dynamically by the processor hardware, and they are associated with the values needed by instructions at various points in time. When a new register value is created (i.e., when an instruction executes that has a register as a destination operand), a new register is allocated for that value. Subsequent instructions that access that value as a source operand in that register must go through a renaming process: the register references in those instructions must be revised to refer to the register containing the needed value. Thus, the same original register reference in several different instructions may refer to different actual registers, if different values are intended.

Let us consider how register renaming could be used on the code fragment we have been examining:

```
I1: R3b ← R3a op R5a
I2: R4b ← R3b + 1
I3: R3c ← R5a + 1
I4: R7b ← R3c op R4b
```

The register reference without the subscript refers to the logical register reference found in the instruction. The register reference with the subscript refers to a hardware register allocated to hold a new value. When a new allocation is made for a particular logical register, subsequent instruction references to that logical register as a source operand are made to refer to the most recently allocated hardware register (recent in terms of the program sequence of instructions).

In this example, the creation of register $R3_c$ in instruction I3 avoids the WAR dependency on the second instruction and the WAW on the first instruction, and it does not interfere with the correct value being accessed by I4. The result is that I3 can be issued immediately; without renaming, I3 cannot be issued until the first instruction is complete and the second instruction is issued.

Scoreboarding Simulator

An alternative to register renaming is a scoreboarding. In essence, scoreboarding is a bookkeeping technique that allows instructions to execute whenever they are not dependent on previous instructions and no structural hazards are present. See Appendix N for a discussion.

Machine Parallelism

In the preceding discussion, we looked at three hardware techniques that can be used in a superscalar processor to enhance performance: duplication of resources, out-of-order issue, and renaming. One study that illuminates the relationship among these techniques was reported in [SMIT89]. The study made use of a simulation that modeled a machine with the characteristics of the MIPS R2000, augmented with various superscalar features. A number of different program sequences were simulated.

Figure 16.6 shows the results. In each of the graphs, the vertical axis corresponds to the mean speedup of the superscalar machine over the scalar machine. The horizontal axis shows the results for four alternative processor organizations. The base machine does not duplicate any of the functional units, but it can issue instructions out of order. The second configuration duplicates the load/store functional unit that accesses a data cache. The third configuration duplicates the ALU, and the fourth configuration duplicates both load/store and ALU. In each graph, results are shown for instruction window sizes of 8, 16, and 32 instructions, which dictates the amount of lookahead the processor can do. The difference between the two graphs is that, in the second, register renaming is allowed. This is equivalent to saying that the first graph reflects a machine that is limited by all dependencies, whereas the second graph corresponds to a machine that is limited only by true dependencies.

The two graphs, combined, yield some important conclusions. The first is that it is probably not worthwhile to add functional units without register renaming. There

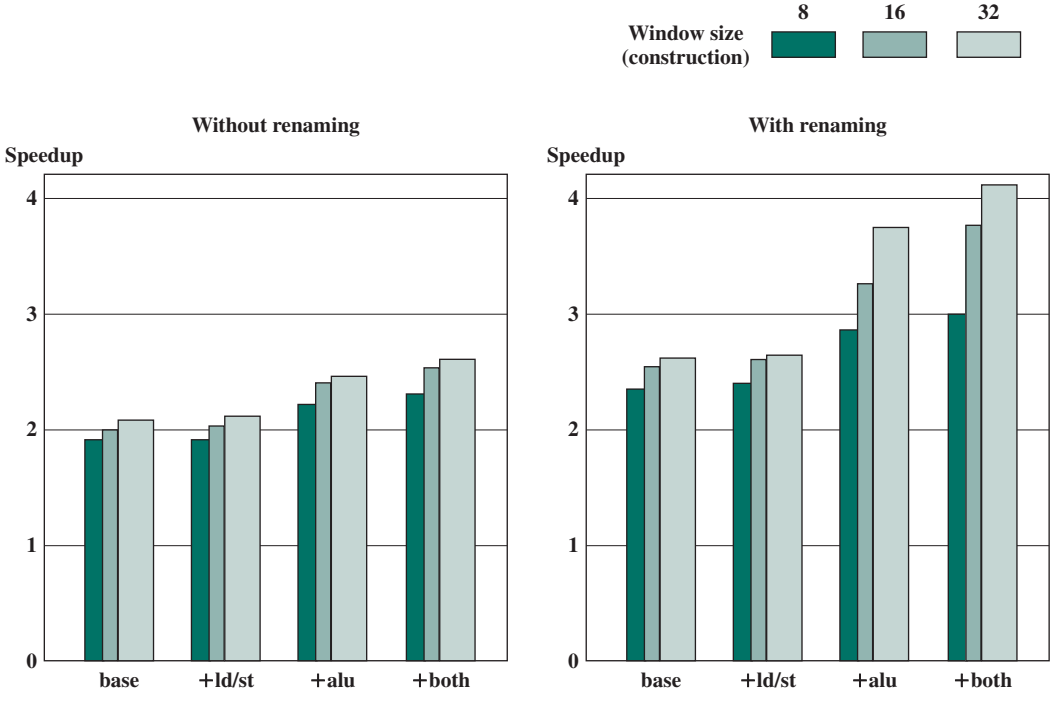


Figure 16.6 Speedups of Various Machine Organizations without Procedural Dependencies

is some slight improvement in performance, but at the cost of increased hardware complexity. With register renaming, which eliminates antidependencies and output dependencies, noticeable gains are achieved by adding more functional units. Note, however, that there is a significant difference in the amount of gain achievable between using an instruction window of 8 versus a larger instruction window. This indicates that if the instruction window is too small, data dependencies will prevent effective utilization of the extra functional units; the processor must be able to look quite far ahead to find independent instructions to utilize the hardware more fully.

Pipeline with Static vs. Dynamic Scheduling—Simulator

Branch Prediction

Any high-performance pipelined machine must address the issue of dealing with branches. For example, the Intel 80486 addressed the problem by fetching both the next sequential instruction after a branch and speculatively fetching the branch target instruction. However, because there are two pipeline stages between prefetch and execution, this strategy incurs a two-cycle delay when the branch gets taken.

With the advent of RISC machines, the delayed branch strategy was explored. This allows the processor to calculate the result of conditional branch instructions before any unusable instructions have been prefetched. With this method, the processor always executes the single instruction that immediately follows the branch. This keeps the pipeline full while the processor fetches a new instruction stream.

With the development of superscalar machines, the delayed branch strategy has less appeal. The reason is that multiple instructions need to execute in the delay slot, raising several problems relating to instruction dependencies. Thus, superscalar machines have returned to pre-RISC techniques of **branch prediction**. Some, like the PowerPC 601, use a simple static branch prediction technique. More sophisticated processors, such as the PowerPC 620 and the Pentium 4, use dynamic branch prediction based on branch history analysis.

Superscalar Execution

We are now in a position to provide an overview of superscalar execution of programs; this is illustrated in Figure 16.7. The program to be executed consists of a linear sequence of instructions. This is the static program as written by the programmer or generated by the compiler. The instruction fetch stage, which includes branch prediction, is used to form a dynamic stream of instructions. This stream is examined for dependencies, and the processor may remove artificial dependencies. The processor then dispatches the instructions into a window of execution. In this window, instructions no longer form a sequential stream but are structured according to their true data dependencies. The processor executes each instruction in an order determined by the true data dependencies and hardware resource availability. Finally, instructions are conceptually put back into sequential order and their results are recorded.

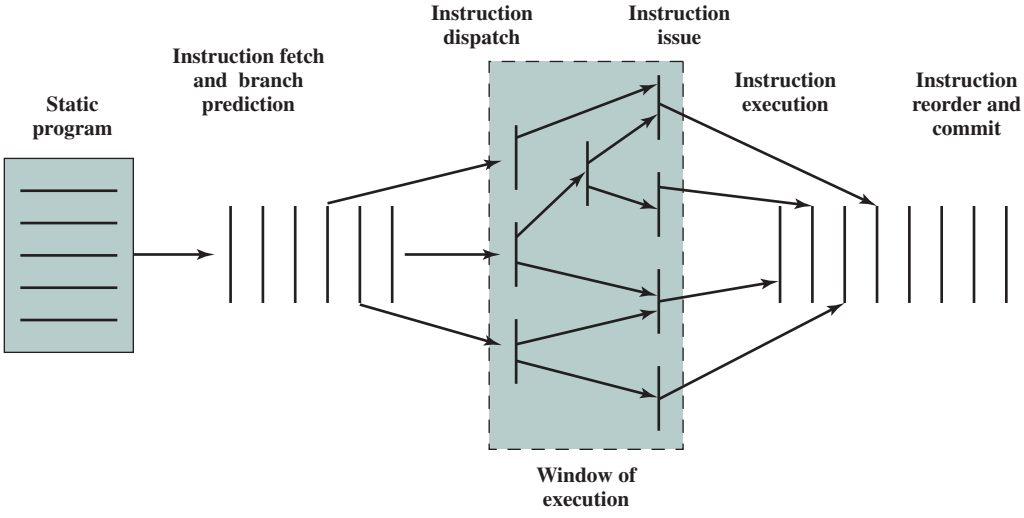


Figure 16.7 Conceptual Depiction of Superscalar Processing

The final step mentioned in the preceding paragraph is referred to as **committing**, or **retiring**, the instruction. This step is needed for the following reason. Because of the use of parallel, multiple pipelines, instructions may complete in an order different from that shown in the static program. Further, the use of branch prediction and speculative execution means that some instructions may complete execution and then must be abandoned because the branch they represent is not taken. Therefore, permanent storage and program-visible registers cannot be updated immediately when instructions complete execution. Results must be held in some sort of temporary storage that is usable by dependent instructions and then made permanent when it is determined that the sequential model would have executed the instruction.

Superscalar Implementation

Based on our discussion so far, we can make some general comments about the processor hardware required for the superscalar approach. [SMIT95] lists the following key elements:

- Instruction fetch strategies that simultaneously fetch multiple instructions, often by predicting the outcomes of, and fetching beyond, conditional branch instructions. These functions require the use of multiple pipeline fetch and decode stages, and branch prediction logic.
- Logic for determining true dependencies involving register values, and mechanisms for communicating these values to where they are needed during execution.
- Mechanisms for initiating, or issuing, multiple instructions in parallel.
- Resources for parallel execution of multiple instructions, including multiple pipelined functional units and memory hierarchies capable of simultaneously servicing multiple memory references.
- Mechanisms for committing the process state in correct order.

16.3 INTEL CORE MICROARCHITECTURE

Although the concept of superscalar design is generally associated with the RISC architecture, the same superscalar principles can be applied to a CISC machine. Perhaps the most notable example of this is the Intel x86 architecture. The evolution of superscalar concepts in the Intel line is interesting to note. The 386 is a traditional CISC nonpipelined machine. The 486 introduced the first pipelined x86 processor, reducing the average latency of integer operations from between two and four cycles to one cycle, but still limited to executing a single instruction each cycle, with no superscalar elements. The original Pentium had a modest superscalar component, consisting of the use of two separate integer execution units. The Pentium Pro introduced a full-blown superscalar design with out-of-order execution. Subsequent x86 models have refined and enhanced the superscalar design.

Figure 16.8 shows the current version of the x86 pipeline architecture. Intel refers to a pipeline architecture as a *microarchitecture*. The microarchitecture

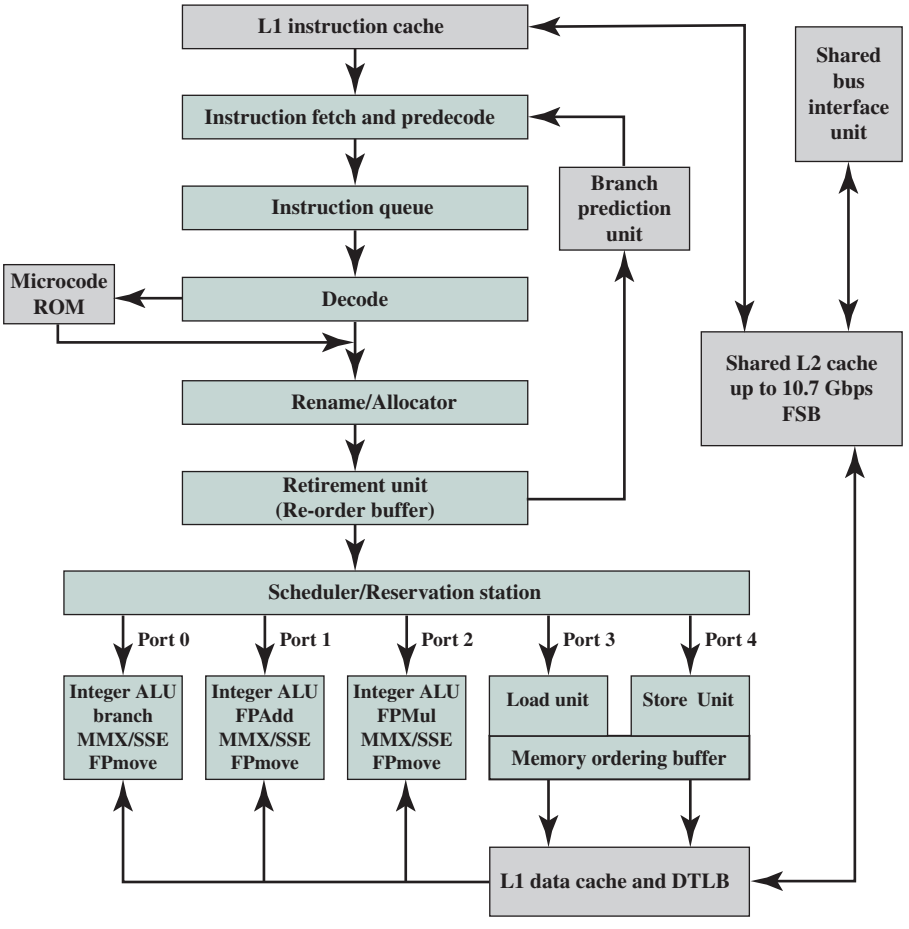


Figure 16.8 Intel Core Microarchitecture

underlies and implements the machine's instruction set architecture. The microarchitecture is referred to as the Intel Core Microarchitecture. It is implemented on each processor core in the Intel Core 2 and Intel Xeon processor families. There is also an Enhanced Intel Core Microarchitecture. One key difference between the two microarchitectures is that the Enhanced Intel Core Microarchitecture provides a third level of cache.

Table 16.2 shows some of the parameters and performance characteristics of the cache architecture. All of the caches use a writeback update policy. When an instruction reads data from a memory location, the processor looks for the cache line that contains this data in the caches and main memory in the following order:

1. L1 data cache of the initiating core
2. L1 data cache of other cores and L2 cache
3. System memory

The cache line is taken from the L1 data cache of another core only if it is modified, ignoring the cache line availability or state in the L2 cache. Table 16.2b

Table 16.2 Cache/Memory Parameters and Performance of Processors Based on Intel Core Microarchitecture

(a) Cache Parameters

Cache Level	Capacity	Associativity (ways)	Line Size (bytes)	Writeback Update Policy
L1 data	32 kB	8	64	Writeback
L1 instruction	32 kB	8	N/A	N/A
L2 (shared)[1]	2, 4 MB	8 or 16	64	Writeback
L2 (shared)[2]	3, 6 MB	12 or 24	64	Writeback
L3 (shared)[2]	8, 12, 16 MB	15	64	Writeback

Notes:
1. Intel Core Microarchitecture
2. Enhanced Intel Core Microarchitecture

(b) Load/Store Performance

Data Locality	Load		Store	
	Latency	Throughput	Latency	Throughput
L1 data cache	3 clock cycles	1 clock cycle	2 clock cycles	3 clock cycles
L1 data cache of the other core in modified state	14 clock cycles + 5.5 bus cycles	14 clock cycles + 5.5 bus cycles	14 clock cycles + 5.5 bus cycles	N/A
L2 cache	14	3	14	3
Memory	14 clock cycles + 5.5 bus cycles + memory latency	Depends on bus read protocol	14 clock cycles + 5.5 bus cycles + memory latency	Depends on bus read protocol

shows the characteristics of fetching the first four bytes of different localities from the memory cluster. The latency column provides an estimate of access latency. However, the actual latency can vary depending on the load of cache, memory components, and their parameters.

The pipeline of the Intel Core microarchitecture contains:

- An in-order issue front end that fetches instruction streams from memory, with four instruction decoders to supply decoded instructions to the out-of-order execution core. Each instruction is translated into one or more fixed-length RISC instructions, known as **micro-operations**, or **micro-ops**.
- An out-of-order superscalar execution core that can issue up to six micro-ops per cycle and reorder micro-ops to execute as soon as sources are ready and execution resources are available.
- An in-order retirement unit that ensures the results of execution of micro-ops are processed and architectural states and the processor's register set are updated according to the original program order.

In effect, the Intel Core Microarchitecture implements a CISC instruction set architecture on a RISC microarchitecture. The inner RISC micro-ops pass through a pipeline with at least 14 stages; in some cases, the micro-op requires multiple execution stages, resulting in an even longer pipeline. This contrasts with the five-stage pipeline (Figure 14.21) used on the earlier Intel x86 processors and on the Pentium.

Front End

The front end needs to supply decoded instructions (micro-ops) and sustain the stream to a six-issue wide out-of-order engine. It consists of three major components: branch prediction unit (BPU), instruction fetch and predecode unit, and instruction queue and decode unit.

BRANCH PREDICTION UNIT This unit helps the instruction fetch unit fetch the most likely instruction to be executed by predicting the various branch types: conditional, indirect, direct, call, and return. The BPU uses dedicated hardware for each branch type. Branch prediction enables the processor to begin executing instructions long before the branch outcome is decided.

The microarchitecture uses a dynamic branch prediction strategy based on the history of recent executions of branch instructions. A branch target buffer (BTB) is maintained that caches information about recently encountered branch instructions. Whenever a branch instruction is encountered in the instruction stream, the BTB is checked. If an entry already exists in the BTB, then the instruction unit is guided by the history information for that entry in determining whether to predict that the branch is taken. If a branch is predicted, then the branch destination address associated with this entry is used for prefetching the branch target instruction.

Once the instruction is executed, the history portion of the appropriate entry is updated to reflect the result of the branch instruction. If this instruction is not represented in the BTB, then the address of this instruction is loaded into an entry in the BTB; if necessary, an older entry is deleted.

The description of the preceding two paragraphs fits, in general terms, the branch prediction strategy used on the original Pentium model, as well as the later

Pentium models, including current Intel models. However, in the case of the Pentium, a relatively simple 2-bit history scheme is used. The later models have much longer pipelines (14 stages for the Intel Core Microarchitecture compared with 5 stages for the Pentium) and therefore the penalty for misprediction is greater. Accordingly, the later models use a more elaborate branch prediction scheme with more history bits to reduce the misprediction rate.

Conditional branches that do not have a history in the BTB are predicted using a static prediction algorithm, according to the following rules:

- For branch addresses that are not instruction pointer (IP) relative, predict taken if the branch is a return and not taken otherwise.
- For IP-relative backward conditional branches, predict taken. This rule reflects the typical behavior of loops.
- For IP-relative forward conditional branches, predict not taken.

INSTRUCTION FETCH AND PREDECODE UNIT The instruction fetch unit comprises the instruction translation lookaside buffer (ITLB), an instruction prefetcher, the instruction cache, and the predecode logic.

Instruction fetch is performed from an L1 instruction cache. When an L1 cache miss occurs, the in-order front end feeds new instructions into the L1 cache from the L2 cache 64 bytes at a time. As a default, instructions are fetched sequentially, so that each L2 cache line fetch includes the next instruction to be fetched. Branch prediction via the branch prediction unit may alter this sequential fetch operation. The ITLB translates the linear IP address given it into physical addresses needed to access the L2 cache. Static branch prediction in the front end is used to determine which instructions to fetch next.

The predecode unit accepts the sixteen bytes from the instruction cache or prefetch buffers and carries out the following tasks:

- Determine the length of the instructions.
- Decode all prefixes associated with instructions.
- Mark various properties of instructions for the decoders (for example, “is branch”).

The predecode unit can write up to six instructions per cycle into the instruction queue. If a fetch contains more than six instructions, the predecoder continues to decode up to six instructions per cycle until all instructions in the fetch are written to the instruction queue. Subsequent fetches can only enter predecoding after the current fetch completes.

INSTRUCTION QUEUE AND DECODE UNIT Fetched instructions are placed in an instruction queue. From there, the decode unit scans the bytes to determine instruction boundaries; this is a necessary operation because of the variable length of x86 instructions. The decoder translates each machine instruction into from one to four micro-ops, each of which is a 118-bit RISC instruction. Note for comparison that most pure RISC machines have an instruction length of just 32 bits. The longer micro-op length is required to accommodate the more complex x86 instructions. Nevertheless, the micro-ops are easier to manage than the original instructions from which they derive.

A few instructions require more than four micro-ops. These instructions are transferred to microcode ROM, which contains the series of micro-ops (five or more) associated with a complex machine instruction. For example, a string instruction may translate into a very large (even hundreds), repetitive sequence of micro-ops. Thus, the microcode ROM is a microprogrammed control unit in the sense discussed in Part Six.

The resulting micro-op sequence is delivered to the rename/allocator module.

Out-of-Order Execution Logic

This part of the processor reorders micro-ops to allow them to execute as quickly as their input operands are ready.

ALLOCATE The allocate stage allocates resources required for execution. It performs the following functions:

- If a needed resource, such as a register, is unavailable for one of the three micro-ops arriving at the allocator during a clock cycle, the allocator stalls the pipeline.
- The allocator allocates a reorder buffer (ROB) entry, which tracks the completion status of one of the 126 micro-ops that could be in process at any time.[2]
- The allocator allocates one of the 128 integer or floating-point register entries for the result data value of the micro-op, and possibly a load or store buffer used to track one of the 48 loads or 24 stores in the machine pipeline.
- The allocator allocates an entry in one of the two micro-op queues in front of the instruction schedulers.

The ROB is a circular buffer that can hold up to 126 micro-ops and also contains the 128 hardware registers. Each buffer entry consists of the following fields:

- **State:** Indicates whether this micro-op is scheduled for execution, has been dispatched for execution, or has completed execution and is ready for retirement.
- **Memory Address:** The address of the Pentium instruction that generated the micro-op.
- **Micro-op:** The actual operation.
- **Alias Register:** If the micro-op references one of the 16 architectural registers, this entry redirects that reference to one of the 128 hardware registers.

Micro-ops enter the ROB in order. Micro-ops are then dispatched from the ROB to the Dispatch/Execute unit out of order. The criterion for dispatch is that the appropriate execution unit and all necessary data items required for this microop are available. Finally, micro-ops are retired from the ROB in order. To accomplish in-order retirement, micro-ops are retired oldest first after each micro-op has been designated as ready for retirement.

REGISTER RENAMING The rename stage remaps references to the 16 architectural registers (8 floating-point registers, plus EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP) into a set of 128 physical registers. The stage removes false dependencies

[2]See Appendix N for a discussion of reorder buffers.

caused by a limited number of architectural registers while preserving the true data dependencies (reads after writes).

MICRO-OP QUEUING After resource allocation and register renaming, micro-ops are placed in one of two micro-op queues, where they are held until there is room in the schedulers. One of the two queues is for memory operations (loads and stores) and the other for micro-ops that do not involve memory references. Each queue obeys a FIFO (first-in-first-out) discipline, but no order is maintained between queues. That is, a micro-op may be read out of one queue out of order with respect to micro-ops in the other queue. This provides greater flexibility to the schedulers.

MICRO-OP SCHEDULING AND DISPATCHING The schedulers are responsible for retrieving micro-ops from the micro-op queues and dispatching these for execution. Each scheduler looks for micro-ops in whose status indicates that the micro-op has all of its operands. If the execution unit needed by that micro-op is available, then the scheduler fetches the micro-op and dispatches it to the appropriate execution unit. Up to six micro-ops can be dispatched in one cycle. If more than one micro-op is available for a given execution unit, then the scheduler dispatches them in sequence from the queue. This is a sort of FIFO discipline that favors in-order execution, but by this time the instruction stream has been so rearranged by dependencies and branches that it is substantially out of order.

Four ports attach the schedulers to the execution units. Port 0 is used for both integer and floating-point instructions, with the exception of simple integer operations and the handling of branch mispredictions, which are allocated to Port 1. In addition, MMX execution units are allocated between these two ports. The remaining ports are for memory loads and stores.

Integer and Floating-Point Execution Units

The integer and floating-point register files are the source for pending operations by the execution units. The execution units retrieve values from the register files as well as from the L1 data cache. A separate pipeline stage is used to compute flags (e.g., zero, negative); these are typically the input to a branch instruction.

A subsequent pipeline stage performs branch checking. This function compares the actual branch result with the prediction. If a branch prediction turns out to have been wrong, then there are micro-operations in various stages of processing that must be removed from the pipeline. The proper branch destination is then provided to the Branch Predictor during a drive stage, which restarts the whole pipeline from the new target address.

16.4 ARM CORTEX-A8

Recent implementations of the ARM architecture have seen the introduction of superscalar techniques in the instruction pipeline. In this section, we focus on the ARM Cortex-A8, which provides a good example of a RISC-based superscalar design.

The Cortex-A8 is in the ARM family of processors that ARM refers to as application processors. An ARM application processor is an embedded processor running complex operating systems for wireless, consumer and imaging applications. The Cortex-A8 targets a wide variety of mobile and consumer applications including mobile phones, set-top boxes, gaming consoles and automotive navigation/entertainment systems.

Figure 16.9 shows a logical view of the Cortex-A8 architecture, emphasizing the flow of instructions among functional units. The main instruction flow is through three functional units that implement a dual, in-order-issue, 13-stage pipeline. The Cortex designers decided to stay with in-order issue to keep additional

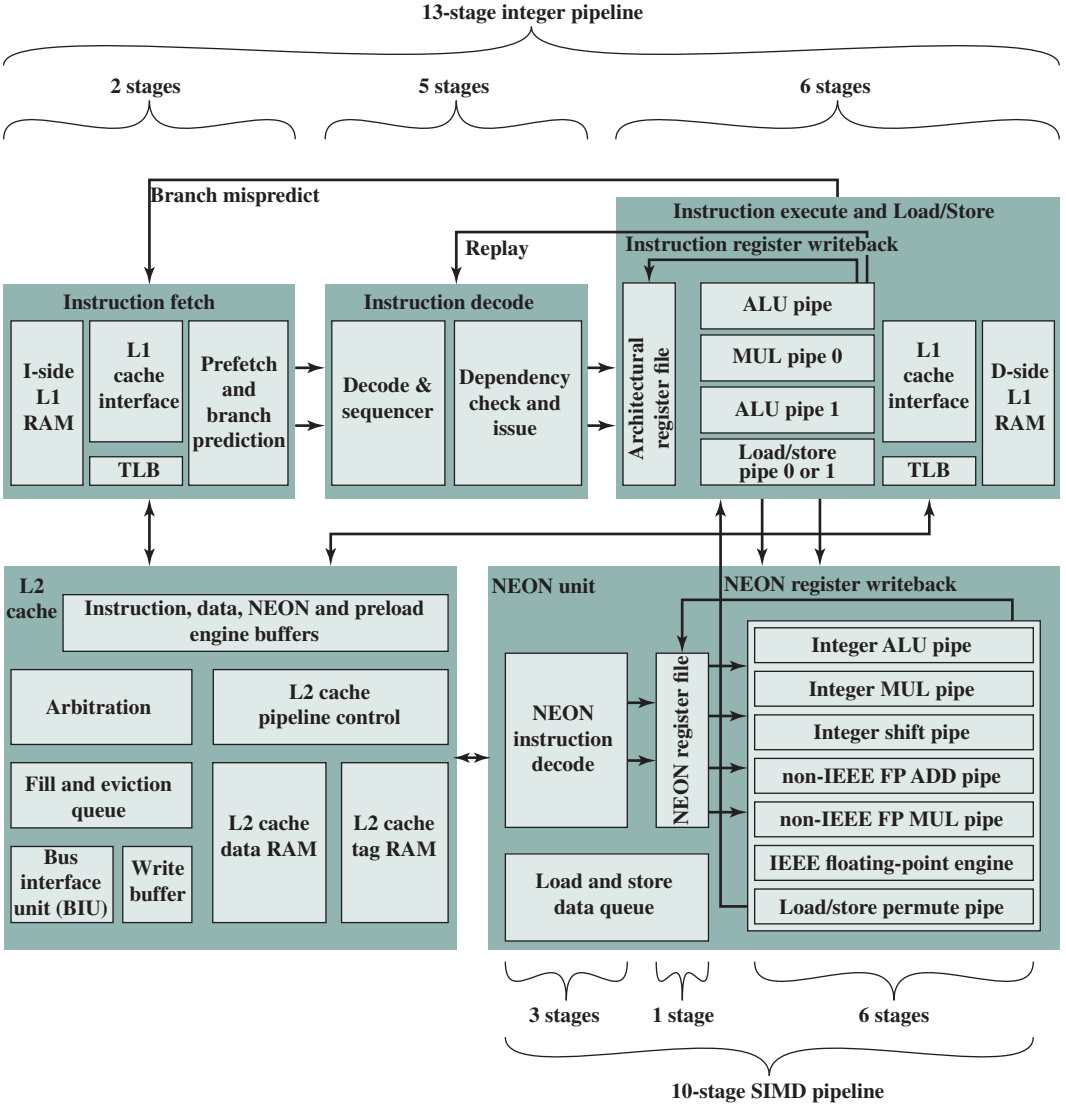


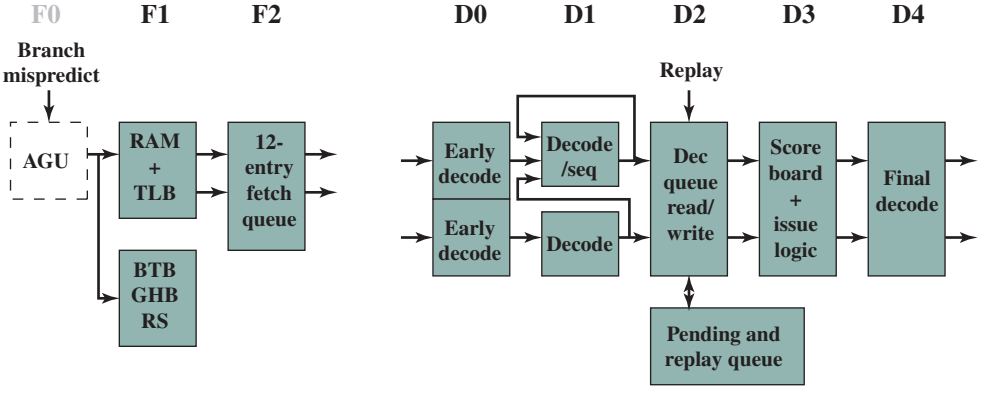
Figure 16.9 Architectural Block Diagram of ARM Cortex-A8

power required to a minimum. Out-of-order issue and **retire** can require extensive amounts of logic consuming extra power.

Figure 16.10 shows the details of the main Cortex-A8 pipeline. There is a separate unit for SIMD (single-instruction-multiple-data) unit that implements a 10-stage pipeline.

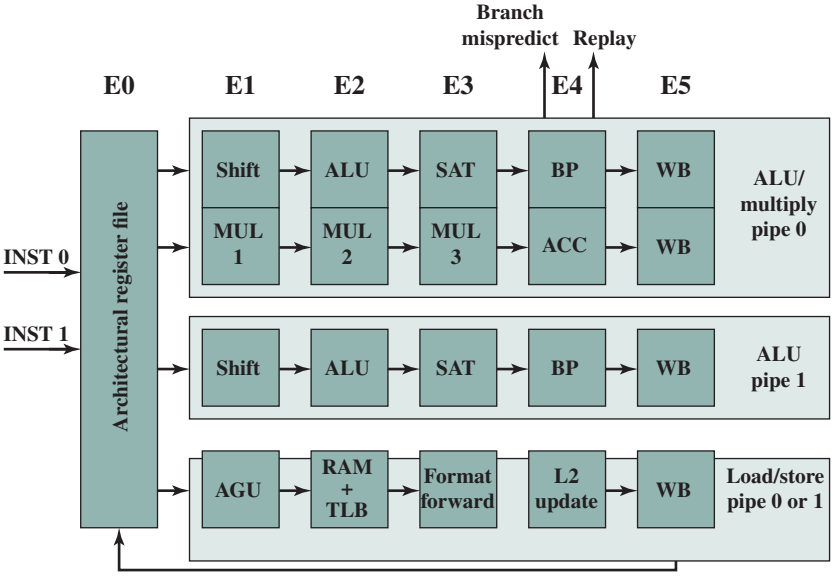
Instruction Fetch Unit

The instruction fetch unit predicts the instruction stream, fetches instructions from the L1 instruction cache, and places the fetched instructions into a buffer for consumption by the decode pipeline. The instruction fetch unit also includes the L1



(a) Instruction fetch pipeline

(b) Instruction decode pipeline



(c) Instruction execute and load/store pipeline

Figure 16.10 ARM Cortex-A8 Integer Pipeline

instruction cache. Because there can be several unresolved branches in the pipeline, instruction fetches are speculative, meaning there is no guarantee that they are executed. A branch or exceptional instruction in the code stream can cause a pipeline flush, discarding the currently fetched instructions. The instruction fetch unit can fetch up to four instructions per cycle, and goes through the following stages:

F0: The address generation unit (AGU) generates a new virtual address. Normally, this address is the next address sequentially from the preceding fetch address. The address can also be a branch target address provided by a branch prediction for a previous instruction. F0 is not counted as part of the 13-stage pipeline, because ARM processors have traditionally defined instruction cache access as the first stage.

F1: The calculated address is used to fetch instructions from the L1 instruction cache. In parallel, the fetch address is used to access the branch prediction arrays to determine if the next fetch address should be based on a branch prediction.

F3: Instruction data are placed into the instruction queue. If an instruction results in branch prediction, the new target address is sent to the address generation unit.

To minimize the branch penalties typically associated with a deeper pipeline, the Cortex-A8 processor implements a two-level global history branch predictor, consisting of the branch target buffer (BTB) and the global history buffer (GHB). These data structures are accessed in parallel with instruction fetches. The BTB indicates whether or not the current fetch address will return a branch instruction and its branch target address. It contains 512 entries. On a hit in the BTB a branch is predicted and the GHB is accessed. The GHB consists of 4096 2-bit counters that encode the strength and direction information of branches. The GHB is indexed by 10-bit history of the direction of the last ten branches encountered and 4 bits of the PC. In addition to the dynamic branch predictor, a return stack is used to predict subroutine return addresses. The return stack has eight 32-bit entries that store the link register value in r14 and the ARM or Thumb state of the calling function. When a return-type instruction is predicted taken, the return stack provides the last pushed address and state.

The instruction fetch unit can fetch and queue up to 12 instructions. It issues instructions to the decode unit two at a time. The queue enables the instruction fetch unit to prefetch ahead of the rest of the integer pipeline and build up a backlog of instructions ready for decoding.

Instruction Decode Unit

The instruction decode unit decodes and sequences all ARM and Thumb instructions. It has a dual pipeline structure, called *pipe0* and *pipe1*, so that two instructions can progress through the unit at a time. When two instructions are issued from the instruction decode pipeline, pipe0 will always contain the older instruction in program order. This means that if the instruction in pipe0 cannot issue, then the instruction in pipe1 will not issue. All issued instructions progress in order down the execution pipeline with results written back into the register file at the end of the execution pipeline. This in-order instruction issue and retire prevents WAR hazards and keeps tracking of WAW

hazards and recovery from flush conditions straightforward. Thus, the main concern of the instruction decode pipeline is the prevention of RAW hazards.

Each instruction goes through five stages of processing.

D0: Thumb instructions are decompressed into 32-bit ARM instructions. A preliminary decode function is performed.

D1: The instruction decode function is completed.

D2: This stage writes instructions into and read instructions from the pending/replay queue structure.

D3: This stage contains the instruction scheduling logic. A scoreboard predicts register availability using static scheduling techniques.[3] Hazard checking is also done at this stage.

D4: Performs the final decode for all the control signals required by the integer execute and load/store units.

In the first two stages, the instruction type, the source and destination operands, and resource requirements for the instruction are determined. A few less commonly used instructions are referred to as multicycle instructions. The D1 stage breaks these instructions down into multiple instruction opcodes that are sequenced individually through the execution pipeline.

The pending queue serves two purposes. First, it prevents a stall signal from D3 from rippling any further up the pipeline. Second, by buffering instructions, there should always be two instructions available for the dual pipeline. In the case where only one instruction is issued, the pending queue enables two instructions to proceed down the pipeline together, even if they were originally sent from the fetch unit in different cycles.

The replay operation is designed to deal with the effects of the memory system on instruction timing. Instructions are statically scheduled in the D3 stage based on a prediction of when the source operand will be available. Any stall from the memory system can result in the minimum of an 8-cycle delay. This 8-cycle delay minimum is balanced with the minimum number of possible cycles to receive data from the L2 cache in the case of an L1 load miss. Table 16.3 gives the most common cases that can result in an instruction replay because of a memory system stall.

To deal with these stalls, a recovery mechanism is used to flush all subsequent instructions in the execution pipeline and reissue (replay) them. To support replay, instructions are copied into the replay queue before they are issued and removed as they write back their results and retire. If a replay signal is issued instructions are retrieved from the replay queue and reenter the pipeline.

The decode unit issues two instructions in parallel to the execution unit, unless it encounters an issue restriction. Table 16.4 shows the most common restriction cases.

Integer Execute Unit

The instruction execute unit consists of two symmetric arithmetic logic unit (ALU) pipelines, an address generator for load and store instructions, and the multiply pipeline. The execute pipelines also perform register write back. The instruction execute unit:

[3]See Appendix N for a discussion of scoreboarding.

Table 16.3 Cortex-A8 Memory System Effects on Instruction Timings

Replay Event	Delay	Description
Load data miss	8 cycles	**1.** A load instruction misses in the L1 data cache. **2.** A request is then made to the L2 data cache. **3.** If a miss also occurs in the L2 data cache, then a second replay occurs. The number of stall cycles depends on the external system memory timing. The minimum time required to receive the critical word for an L2 cache miss is approximately 25 cycles, but can be much longer because of L3 memory latencies.
Data TLB miss	24 cycles	**1.** A table walk because of a miss in the L1 TLB causes a 24-cycle delay, assuming the translation table entries are found in the L2 cache. **2.** If the translation table entries are not present in the L2 cache, the number of stall cycles depends on the external system memory timing.
Store buffer full	8 cycles plus latency to drain fill buffer	**1.** A store instruction miss does not result in any stalls unless the store buffer is full. **2.** In the case of a full store buffer, the delay is at least eight cycles. The delay can be more if it takes longer to drain some entries from the store buffer.
Unaligned load or store request	8 cycles	**1.** If a load instruction address is unaligned and the full access is not contained within a 128-bit boundary, there is a 8-cycle penalty. **2.** If a store instruction address is unaligned and the full access is not contained within a 64-bit boundary, there is a 8-cycle penalty.

- Executes all integer ALU and multiply operations, including flag generation.
- Generates the virtual addresses for loads and stores and the base write-back value, when required.
- Supplies formatted data for stores and forwards data and flags.
- Processes branches and other changes of instruction stream and evaluates instruction condition codes.

For ALU instructions, either pipeline can be used, consisting of the following stages:

E0: Access register file. Up to six registers can be read from the register file for two instructions.

E1: The barrel shifter (see Figure 14.25) performs its function, if needed.

E2: The ALU unit (see Figure 14.25) performs its function.

E3: If needed, this stage completes saturation arithmetic used by some ARM data processing instructions.

E4: Any change in control flow, including branch misprediction, exceptions, and memory system replays are prioritized and processed.

E5: Results of ARM instructions are written back into the register file.

Table 16.4 Cortex-A8 Dual-Issue Restrictions

Restriction Type	Description	Example	Cycle	Restriction

Load/store resource hazard	There is only one LS pipeline. Only one LS instruction can be issued per cycle. It can be in pipeline 0 or pipeline 1.	LDR r5, [r6]
		STR r7, [r8]
		MOV r9, r10
			1
			2
			2
				Wait for LS unit
				Dual issue possible

Multiply resource hazard	There is only one multiply pipeline, and it is only available in pipeline 0.	ADD r1, r2, r3
		MUL r4, r5, r6
		MUL r7, r8, r9
			1
			2
			3
				Wait for pipeline 0
				Wait for multiply unit

Branch resource hazard	There can be only one branch per cycle. It can be in pipeline 0 or pipeline 1. A branch is any instruction that changes the PC.	BX r1 BEQ 0x1000 ADD r1, r2, r3	1 2 2	Wait for branch Dual issue possible

Data output hazard	Instructions with the same destination cannot be issued in the same cycle. This can happen with conditional code.	MOVEQ r1, r2
		MOVNE r1, r3
		LDR r5, [r6]
			1
			2
			2
				Wait because of output dependency
				Dual issue possible

Data source hazard	Instructions cannot be issued if their data is not available. See the scheduling tables for source requirements and stages results.	ADD r1, r2, r3
		ADD r4, r1, r6
		LDR r7, [r4]
			1
			2
			4
				Wait for r1
				Wait two cycles for r4

Multi-cycle instructions	Multi-cycle instructions must issue in pipeline 0 and can only dual issue in their last iteration.	MOV r1, r2
		LDM r3, {r4-r7}
		LDM (cycle 2)
		LDM (cycle 3)
		
		ADD r8, r9, r10
			1
			2
			3
			4
			
			4
				Wait for pipeline 0, transfer r4
				Transfer r5, r6
				Transfer r7
				Dual issue possible on last transfer

Instructions that invoke the multiply unit (see Figure 14.25) are routed to pipe0; the multiply operation is performed in stages E1 through E3, and the multiply accumulate operation in stage E4.

The load/store pipeline runs parallel to the integer pipeline. The stages are as follows:

E1: The memory address is generated from the base and index register.

E2: The address is applied to the cache arrays.

E3: In the case of a load, data are returned and formatted for forwarding to the ALU or MUL unit. In the case of a store, the data are formatted and ready to be written into the cache.

E4: Performs updates to the L2 cache, if required.

E5: Results of ARM instructions are written back into the register file.

Table 16.5 shows a sample code segment and indicates how the processor might schedule it.

Table 16.5 Cortex-A8 Example Dual Issue Instruction Sequence for Integer Pipeline

Cycle	Program Counter	Instruction	Timing Description
1	0x00000ed0	BX r14	Dual issue pipeline 0
1	0x00000ee4	CMP r0,#0	Dual issue in pipeline 1
2	0x00000ee8	MOV r3,#3	Dual issue pipeline 0
2	0x00000eec	MOV r0,#0	Dual issue in pipeline 1
3	0x00000ef0	STREQ r3,[r1,#0]	Dual issue in pipeline 0, r3 not needed until E3
3	0x00000ef4	CMP r2,#4	Dual issue in pipeline 1
4	0x00000ef8	LDRLS pc,[pc,r2,LSL #2]	Single issue pipeline 0, + 1 cycle for load to pc, no extra cycle for shift since LSL #2
5	0x00000f2c	MOV r0,#1	Dual issue with 2nd iteration of load in pipeline 1
6	0x00000f30	B {pc} + 8	#0xf38 dual issue pipeline 0
6	0x00000f38	STR r0,[r1,#0]	Dual issue pipeline 1
7	0x00000f3c:	LDR pc,[r13],#4	Single issue pipeline 0, + 1 cycle for load to pc
8	0x0000017c	ADD r2,r4,#0xc	Dual issue with 2nd iteration of load in pipeline 1
9	0x00000180	LDR r0,[r6,#4]	Dual issue pipeline 0
9	0x00000184	MOV r1,#0xa	Dual issue pipeline 1
12	0x00000188	LDR r0,[r0,#0]	Single issue pipeline 0: r0 produced in E3, required in E1, so + 2 cycle stall
13	0x0000018c	STR r0,[r4,#0]	Single issue pipeline 0 due to LS resource hazard, no extra delay for r0 since produced in E3 and consumed in E3
14	0x00000190	LDR r0,[r4,#0xc]	Single issue pipeline 0 due to LS resource hazard
15	0x00000194	LDMFD r13!,{r4-r6,r14}	Load multiple: loads r4 in 1st cycle, r5 and r6 in 2nd cycle, r14 in 3rd cycle, 3 cycles total
17	0x00000198	B {pc} + 0xda8	#0xf40 dual issue in pipeline 1 with 3rd cycle of LDM
18	0x00000f40	ADD r0,r0,#2 ARM	Single issue in pipeline 0
19	0x00000f44	ADD r0,r1,r0 ARM	Single issue in pipeline 0, no dual issue due to hazard on r0 produced in E2 and required in E2

SIMD and Floating-Point Pipeline

All SIMD and floating-point instructions pass through the integer pipeline and are processed in a separate 10-stage pipeline (Figure 16.11). This unit, referred to as the NEON unit, handles packed SIMD instructions, and provides two types of floating-point support. If implemented, a vector floating-point (VFP) coprocessor performs floating-point operations in compliance with IEEE 754. If the coprocessor is not present, then separate multiply and add pipelines implement the floating-point operations.

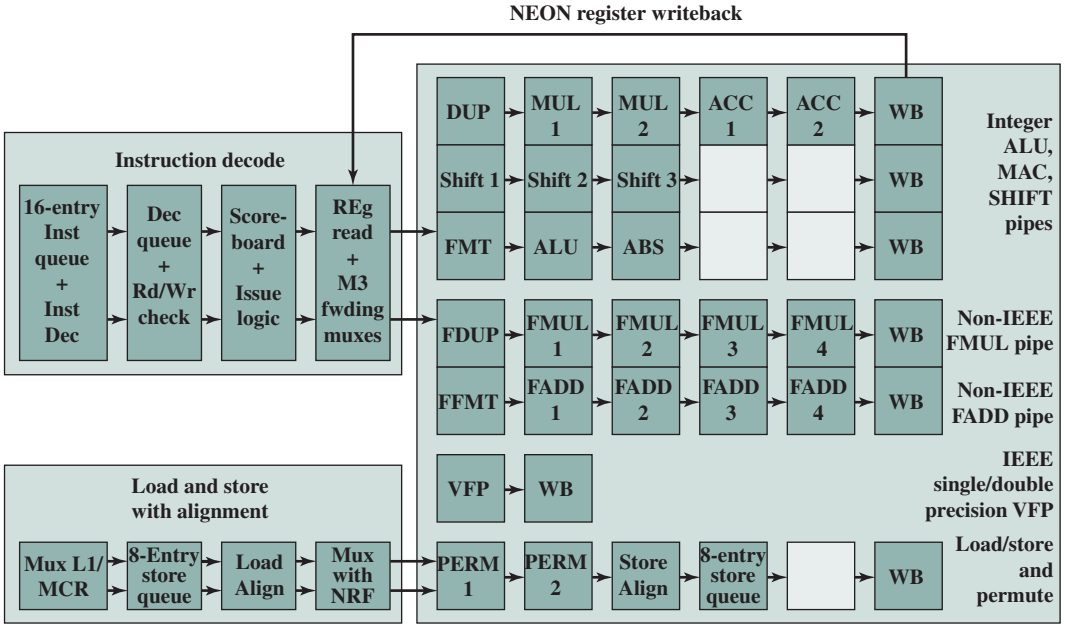


Figure 16.11 ARM Cortex-A8 NEON and Floating-Point Pipeline

16.5 ARM CORTEX-M3

The preceding section looked at the rather complex pipeline organization of the Cortex-A8, an application processor. As a useful contrast, this section examines the considerably simpler pipeline organization of the Cortex-M3. The Cortex-M series is designed for the microcontroller domain. As such, the Cortex-M processors need to be as simple and efficient as possible.

Figure 16.12 provides a block diagram overview of the Cortex-M3 processor. This figure provides more detail than that shown in Figure 1.16. Key elements include:

- **Processor core:** Includes a three-stage pipeline, a register bank, and a memory interface.
- **Memory protection unit:** Protects critical data used by the operating system from user applications, separating processing tasks by disallowing access to each other's data, disabling access to memory regions, allowing memory regions to be defined as read-only, and detecting unexpected memory accesses that could potentially break the system.
- **Nested vectored interrupt controller (NVIC):** Provides configurable interrupt handling abilities to the processor. It facilitates low-latency exception and interrupt handling, and controls power management.
- **Wake-up interrupt controller (NVIC):** Provides configurable interrupt handling abilities to the processor. It facilitates low-latency exception and interrupt handling, and controls power management.
- **Flash patch and breakpoint unit:** Implements breakpoints and code patches.

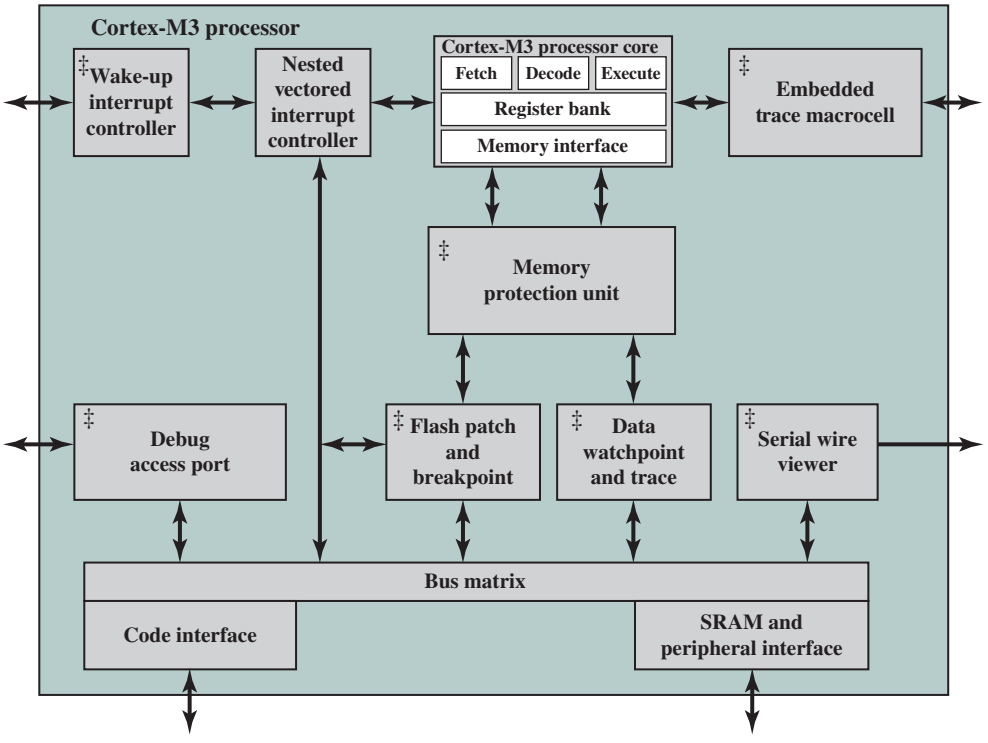


Figure 16.12 ARM Cortex-M3 Block Diagram

- **Data watchpoint and trace (DWT):** Implements watchpoints, data tracing, and system profiling.
- **Serial wire viewer:** Can export a stream of software-generated messages, data trace, and profiling information through a single pin.
- **Debug access port:** Provides an interface for external debug access to the processor.
- **Embedded trace macrocell:** Is an application-driven trace source that supports printf() style debugging to trace operating system and application events, and generates diagnostic system information.
- **Bus matrix:** Connects the core and debug interfaces to external buses on the microcontroller.

Pipeline Structure

The Cortex-M3 pipeline has three stages (Figure 16.12). We examine these in turn.

During the fetch stage, one 32-bit word is fetched at a time and loaded into a 3-word buffer. The 32-bit word may consist of:

- two Thumb instructions,
- one word-aligned Thumb-2 instruction, or

- the upper/lower halfword of a halfword-aligned Thumb-2 instruction with
  - — one Thumb instruction, or
  - — the lower/upper halfword of another halfword-aligned Thumb-2 instruction.

All fetch addresses from the core are word aligned. If a Thumb-2 instruction is halfword aligned, two fetches are necessary to fetch the Thumb-2 instruction. However, the three-entry prefetch buffer ensures that a stall cycle is only necessary for the first halfword Thumb-2 instruction fetched.

This decode stage performs three key functions:

- **Instruction decode and register read:** Decodes Thumb and Thumb-2 instructions.
- **Address generation:** The address generation unit (AGU) generates main memory addresses for the load/store unit.
- **Branch:** Performs branch based on immediate offset in branch instruction or a return based on the contents of the link register (register R14).

Finally, there is a single execute stage for instruction execution, which includes ALU, load/store, and branch instructions.

Dealing with Branches

To keep the processor as simple as possible, the Cortex-M3 processor does not use branch prediction, but instead use the simple techniques of branch forwarding and branch speculation, defined as follows:

- **Branch forwarding:** The term *forwarding* refers to presenting an instruction address to be fetched from memory. The processor forwards certain branch types, by which the memory transaction of the branch is presented at least one cycle earlier than when the opcode reaches execute. Branch forwarding increases the performance of the core, because branches are a significant part of embedded controller applications. Branches affected are PC relative with immediate offset, or use link register (LR) as the target register.
- **Branch speculation:** For conditional branches, the instruction address is presented speculatively, so that the instruction is fetched from memory before it is known if the instruction will be executed.

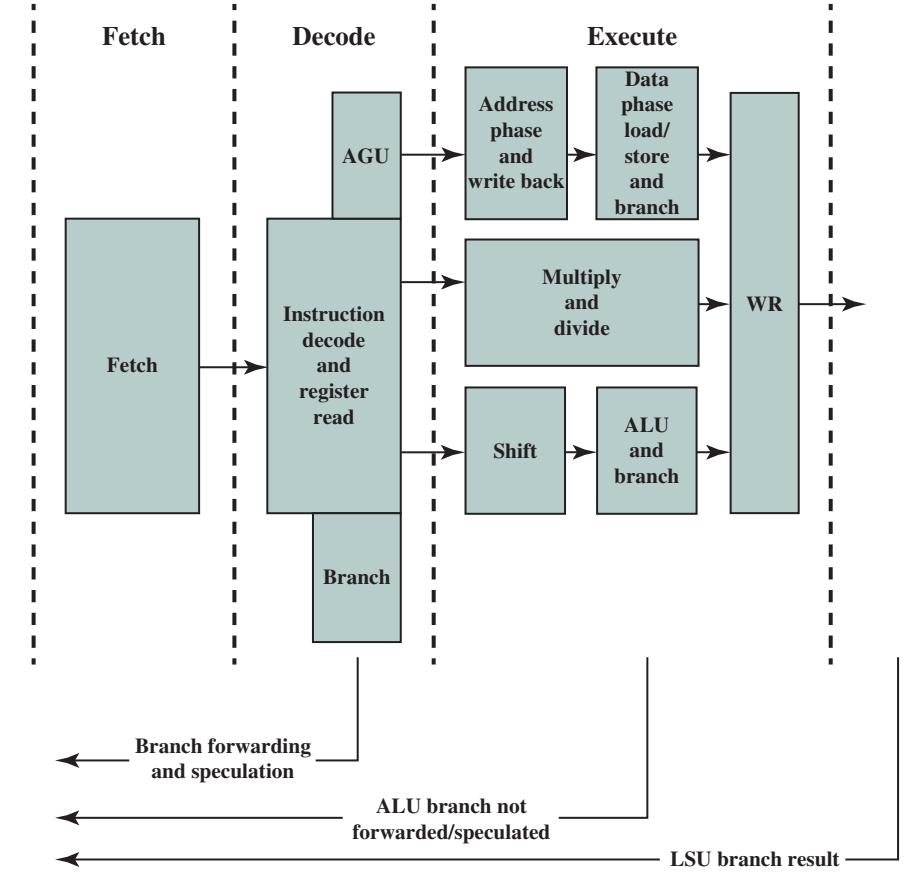
The Cortex-M3 processor prefetches instruction ahead of execution using the fetch buffer. It also speculatively prefetches from branch target addresses. Specifically, when a conditional branch instruction is encountered, the decode stage also includes a speculative instruction fetch that could lead to faster execution. The processor fetches the branch destination instruction during the decode stage itself. Later, during the execute stage, the branch is resolved and it is known which instruction is to be executed next.

If the branch is not to be taken, the next sequential instruction is already available. If the branch is to be taken, the branch instruction is made available at the same time as the decision is made, restricting idle time to just one cycle.

Figure 16.13 clarifies the manner in which branches are handled, which can be described as follows:

1. The decode stage forwards addresses from unconditional branches and speculatively forwards addresses from conditional branches when it is possible to calculate the address.
2. If the ALU determines that a branch is not taken, this information is fed back to empty the instruction cache.
3. A load instruction to the program counter results in a branch address to be forwarded for fetching.

As can be seen, the manner in which branches are handled is considerably simpler for the Cortex-M than the Cortex-A, requiring less processor logic and processing.



AGU = address generation unit

Figure 16.13 ARM Cortex-M3 Pipeline

16.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

antidependency	machine parallelism	register renaming
branch prediction	micro-operations	resource conflict
commit	micro-ops	retire
flow dependency	out-of-order completion	superpipelined
in-order completion	out-of-order issue	superscalar
in-order issue	output dependency	true data dependency
instruction issue	procedural dependency	write-read dependency
instruction-level parallelism	read-write dependency	write-write dependency
instruction window		

Review Questions

16.1 What is the essential characteristic of the superscalar approach to processor design?

16.2 What is the difference between the superscalar and superpipelined approaches?

16.3 What is instruction-level parallelism?

16.4 Briefly define the following terms:
- True data dependency
- Procedural dependency
- Resource conflicts
- Output dependency
- Antidependency

16.5 What is the distinction between instruction-level parallelism and machine parallelism?

16.6 List and briefly define three types of superscalar instruction issue policies.

16.7 What is the purpose of an instruction window?

16.8 What is register renaming and what is its purpose?

16.9 What are the key elements of a superscalar processor organization?

Problems

16.1 When out-of-order completion is used in a superscalar processor, resumption of execution after interrupt processing is complicated, because the exceptional condition may have been detected as an instruction that produced its result out of order. The program cannot be restarted at the instruction following the exceptional instruction, because subsequent instructions have already completed, and doing so would cause these instructions to be executed twice. Suggest a mechanism or mechanisms for dealing with this situation.

16.2 Consider the following sequence of instructions, where the syntax consists of an opcode followed by the destination register followed by one or two source registers:

```
0    ADD     R3, R1, R2
1    LOAD    R6, [R3]
2    AND     R7, R5, 3
3    ADD     R1, R6, R7
4    SRL     R7, R0, 8
```

```
 5    OR      R2, R4, R7
 6    SUB     R5, R3, R4
 7    ADD     R0, R1, 10
 8    LOAD    R6, [R5]
 9    SUB     R2, R1, R6
10    AND     R3, R7, 15
```

Assume the use of a four-stage pipeline: fetch, decode/issue, execute, write back. Assume that all pipeline stages take one clock cycle except for the execute stage. For simple integer arithmetic and logical instructions, the execute stage takes one cycle, but for a LOAD from memory, five cycles are consumed in the execute stage.

If we have a simple scalar pipeline but allow out-of-order execution, we can construct the following table for the execution of the first seven instructions:

Instruction	Fetch	Decode	Execute	Write Back
0	0	1	2	3
1	1	2	4	9
2	2	3	5	6
3	3	4	10	11
4	4	5	6	7
5	5	6	8	10
6	6	7	9	12

The entries under the four pipeline stages indicate the clock cycle at which each instruction begins each phase. In this program, the second ADD instruction (instruction 3) depends on the LOAD instruction (instruction 1) for one of its operands, r6. Because the LOAD instruction takes five clock cycles, and the issue logic encounters the dependent ADD instruction after two clocks, the issue logic must delay the ADD instruction for three clock cycles. With an out-of-order capability, the processor can stall instruction 3 at clock cycle 4, and then move on to issue the following three independent instructions, which enter execution at clocks 6, 8, and 9. The LOAD finishes execution at clock 9, and so the dependent ADD can be launched into execution on clock 10.

a. Complete the preceding table.

b. Redo the table assuming no out-of-order capability. What is the savings using the capability?

c. Redo the table assuming a superscalar implementation that can handle two instructions at a time at each stage.

16.3 Consider the following assembly language program:

```
I1: Move R3, R7          /R3 ← (R7)/
I2: Load R8, (R3)        /R8 ← Memory (R3)/
I3: Add R3, R3, 4        /R3 ← (R3) + 4/
I4: Load R9, (R3)        /R9 ← Memory (R3)/
I5: BLE R8, R9, L3       /Branch if (R9) > (R8)/
```

This program includes WAW, RAW, and WAR dependencies. Show these.

16.4 **a.** Identify the RAW, WAR, and WAW dependencies in the following instruction sequence:

```
I1: R1 = 100
I2: R1 = R2 + R4
I3: R2 = r4 − 25
```

```
I4: R4 = R1 + R3
I5: R1 = R1 + 30
```

b. Rename the registers from part (a) to prevent dependency problems. Identify references to initial register values using the subscript "a" to the register reference.

16.5 Consider the "in-order-issue/in-order-completion" execution sequence shown in Figure 16.14.

a. Identify the most likely reason why I2 could not enter the execute stage until the fourth cycle. Will "in-order issue/out-of-order completion" or "out-of-order issue/out-of-order completion" fix this? If so, which?

b. Identify the reason why I6 could not enter the write stage until the nineth cycle. Will "in-order issue/out-of-order completion" or "out-of-order issue/out-of-order completion" fix this? If so, which?

16.6 Figure 16.15 shows an example of a superscalar processor organization. The processor can issue two instructions per cycle if there is no resource conflict and no data dependence problem. There are essentially two pipelines, with four processing stages (fetch, decode, execute, and store). Each pipeline has its own fetch decode and store unit. Four functional units (multiplier, adder, logic unit, and load unit) are available for use in the execute stage and are shared by the two pipelines on a dynamic basis. The two store units can be dynamically used by the two pipelines, depending on availability at a particular cycle. There is a lookahead window with its own fetch and decoding logic. This window is used for instruction lookahead for out-of-order instruction issue.

Consider the following program to be executed on this processor:

```
I1: Load R1, A     /R1 ← Memory (A)/
I2: Add R2, R1     /R2 ← (R2) + R(1)/
I3: Add R3, R4     /R3 ← (R3) + R(4)/
I4: Mul R4, R5     /R4 ← (R4) + R(5)/
I5: Comp R6        /R6 ← (R6)/
I6: Mul R6, R7     /R6 ← (R6) × R(7)/
```

a. What dependencies exist in the program?

b. Show the pipeline activity for this program on the processor of Figure 16.15 using in-order issue with in-order completion policies and using a presentation similar to Figure 16.2.

b. Repeat for in-order issue with out-of-order completion.

c. Repeat for out-of-order issue with out-of-order completion.

16.7 Figure 16.16 is from a paper on superscalar design. Explain the three parts of the figure, and define w, x, y, and z.

16.8 Yeh's dynamic branch prediction algorithm, used on the Pentium 4, is a two-level branch prediction algorithm. The first level is the history of the last n branches. The

Decode		Execute			Write		Cycle
I1	I2						1
	I2			I1			2
	I2			I1			3
I3	I4		I2				4
I5	I6		I4	I3	I1	I2	5
I5	I6	I5		I3			6
		I5	I6		I3	I4	7
							8
					I5	I6	9

Figure 16.14 An In-Order Issue, In-Order-Completion Execution Sequence

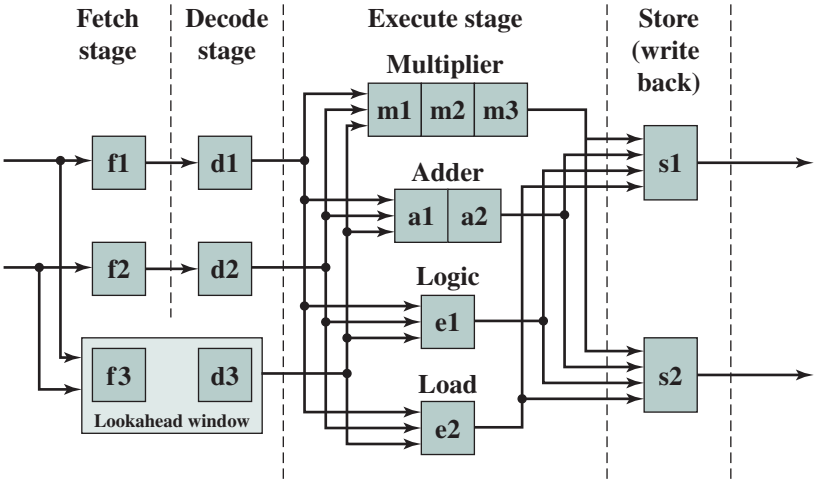


Figure 16.15 A Dual-Pipeline Superscalar Processor

second level is the branch behavior of the last s occurrences of that unique pattern of the last *n* branches. For each conditional branch instruction in a program, there is an entry in a Branch History Table (BHT). Each entry consists of *n* bits corresponding to the last *n* executions of the branch instruction, with a 1 if the branch was taken and a 0 if the branch was not. Each BHT entry indexes into a Pattern Table (PT) that has $2n$ entries, one for each possible pattern of *n* bits. Each PT entry consists of s bits that are used in branch prediction, as was described in Chapter 14 (e.g., Figure 14.19). When a conditional branch is encountered during instruction fetch and decode, the address of the instruction is used to retrieve the appropriate BHT entry, which shows the recent history of the instruction. Then, the BHT entry is used to retrieve the appropriate PT entry for branch prediction. After the branch is executed, the BHT entry is updated, and then the appropriate PT entry is updated.

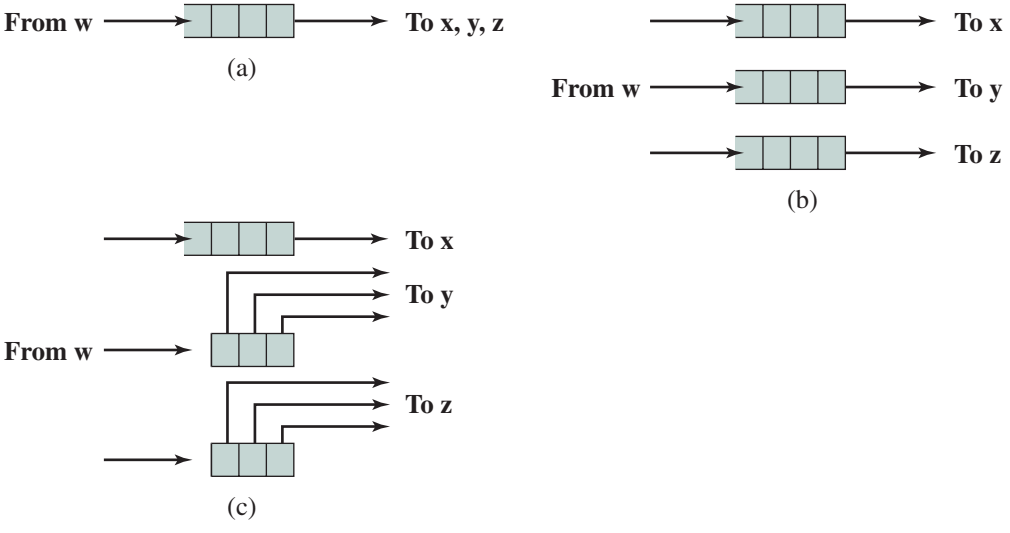


Figure 16.16 Figure for Problem 16.7

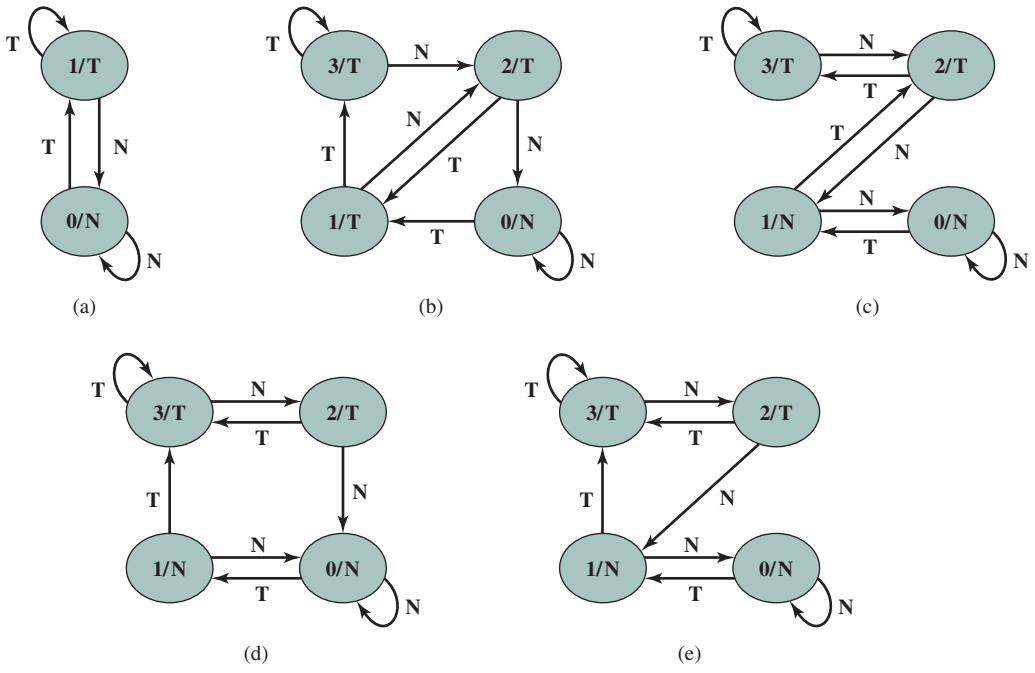


Figure 16.17 Figure for Problem 16.8

a. In testing the performance of this scheme, Yeh tried five different prediction schemes, illustrated in Figure 16.17. Identify which three of these schemes correspond to those shown in Figures 14.19 and 14.28. Describe the remaining two schemes.

b. With this algorithm, the prediction is not based on just the recent history of this particular branch instruction. Rather, it is based on the recent history of all patterns of branches that match the *n*-bit pattern in the BHT entry for this instruction. Suggest a rationale for such a strategy.

PART FIVE PARALLEL ORGANIZATION

CHAPTER 17

PARALLEL PROCESSING

17.1 Multiple Processor Organizations
- Types of Parallel Processor Systems
- Parallel Organizations

17.2 Symmetric Multiprocessors
- Organization
- Multiprocessor Operating System Design Considerations

17.3 Cache Coherence and the MESI Protocol
- Software Solutions
- Hardware Solutions
- The MESI Protocol

17.4 Multithreading and Chip Multiprocessors
- Implicit and Explicit Multithreading
- Approaches to Explicit Multithreading

17.5 Clusters
- Cluster Configurations
- Operating System Design Issues
- Cluster Computer Architecture
- Blade Servers
- Clusters Compared to SMP

17.6 Nonuniform Memory Access
- Motivation
- Organization
- NUMA Pros and Cons

17.7 Cloud Computing
- Cloud Computing Elements
- Cloud Computing Reference Architecture

17.8 Key Terms, Review Questions, and Problems

Learning Objectives

After studying this chapter, you should be able to:

- Summarize the types of parallel **processor organizations**.
- Present an overview of design features of **symmetric multiprocessors**.
- Understand the issue of **cache coherence** in a multiple processor system.
- Explain the key features of the **MESI protocol**.
- Explain the difference between **implicit** and **explicit multithreading**.
- Summarize key design issues for **clusters**.
- Explain the concept of **nonuniform memory** access.
- Present an overview of **cloud computing** concepts

Traditionally, the computer has been viewed as a sequential machine. Most computer programming languages require the programmer to specify algorithms as sequences of instructions. Processors execute programs by executing machine instructions in a sequence and one at a time. Each instruction is executed in a sequence of operations (fetch instruction, fetch operands, perform operation, store results).

This view of the computer has never been entirely true. At the micro-operation level, multiple control signals are generated at the same time. Instruction pipelining, at least to the extent of overlapping fetch and execute operations, has been around for a long time. Both of these are examples of performing independent operations in parallel. This approach is taken further with superscalar organization, which exploits instruction-level parallelism. With a superscalar machine, there are multiple execution units within a single processor, and these may execute multiple instructions from the same program in parallel.

As computer technology has evolved, and as the cost of computer hardware has dropped, computer designers have sought more and more opportunities for parallelism, usually to enhance performance and, in some cases, to increase availability. After an overview, this chapter looks at some of the most prominent approaches to parallel organization. First, we examine symmetric multiprocessors (SMPs), one of the earliest and still the most common example of parallel organization. In an SMP organization, multiple processors share a common memory. This organization raises the issue of cache coherence, to which a separate section is devoted. Next, the chapter examines multithreaded processors and chip multiprocessors. Then we describe clusters, which consist of multiple independent computers organized in a cooperative fashion. Clusters have become increasingly common to support workloads that are beyond the capacity of a single SMP. Another approach to the use of multiple processors that we examine is that of nonuniform memory access (NUMA) machines. The NUMA approach is relatively new and not yet proven in the marketplace, but is often considered as an alternative to the SMP or cluster approach. Finally, this chapter looks at cloud computing architecture.

17.1 MULTIPLE PROCESSOR ORGANIZATIONS

Types of Parallel Processor Systems

A taxonomy first introduced by Flynn [FLYN72] is still the most common way of categorizing systems with parallel processing capability. Flynn proposed the following categories of computer systems:

- **Single instruction, single data (SISD) stream:** A single processor executes a single instruction stream to operate on data stored in a single memory. Uniprocessors fall into this category.
- **Single instruction, multiple data (SIMD) stream:** A single machine instruction controls the simultaneous execution of a number of processing elements on a lockstep basis. Each processing element has an associated data memory, so that instructions are executed on different sets of data by different processors. Vector and array processors fall into this category, and are discussed in Section 18.7.
- **Multiple instruction, single data (MISD) stream:** A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. This structure is not commercially implemented.
- **Multiple instruction, multiple data (MIMD) stream:** A set of processors simultaneously execute different instruction sequences on different data sets. SMPs, clusters, and NUMA systems fit into this category.

With the MIMD organization, the processors are general purpose; each is able to process all of the instructions necessary to perform the appropriate data transformation. MIMDs can be further subdivided by the means in which the processors communicate (Figure 17.1). If the processors share a common memory, then each processor accesses programs and data stored in the shared memory, and processors communicate with each other via that memory. The most common form of such system is known as a **symmetric multiprocessor (SMP)**, which we examine in Section 17.2. In an SMP, multiple processors share a single memory or pool of memory by means of a shared bus or other interconnection mechanism; a distinguishing feature is that the memory access time to any region of memory is approximately the same for each processor. A more recent development is the **nonuniform memory access (NUMA)** organization, which is described in Section 17.5. As the name suggests, the memory access time to different regions of memory may differ for a NUMA processor.

A collection of independent uniprocessors or SMPs may be interconnected to form a **cluster**. Communication among the computers is either via fixed paths or via some network facility.

Parallel Organizations

Figure 17.2 illustrates the general organization of the taxonomy of Figure 17.1. Figure 17.2a shows the structure of an SISD. There is some sort of control unit (CU) that provides an instruction stream (IS) to a processing unit (PU). The processing

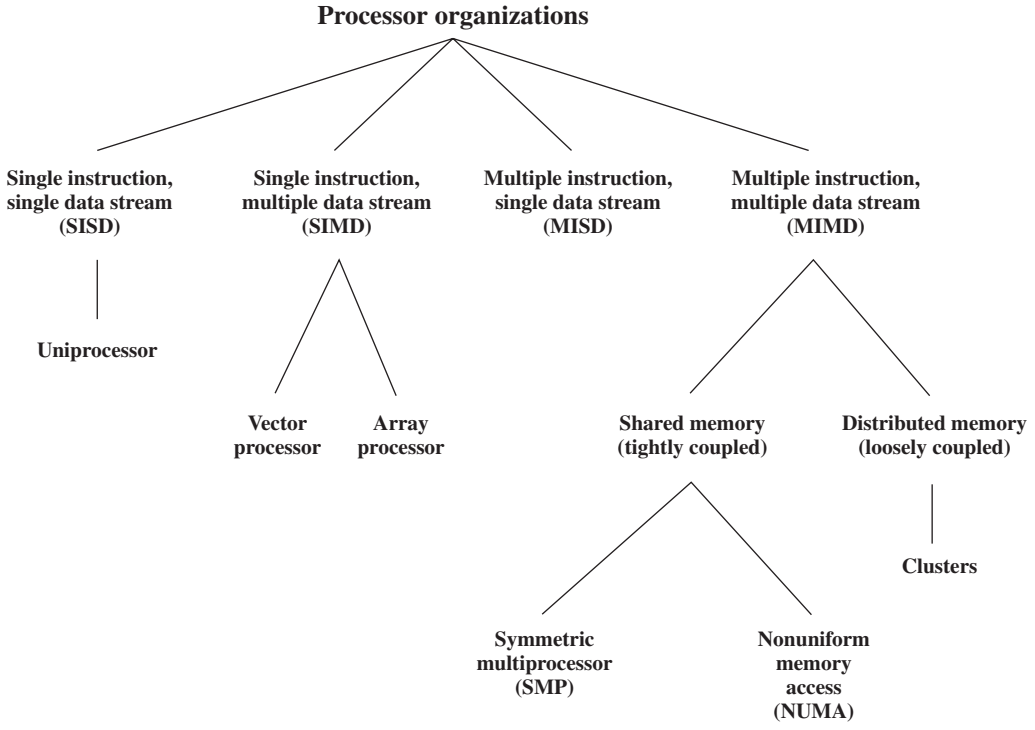


Figure 17.1 A Taxonomy of Parallel Processor Architectures

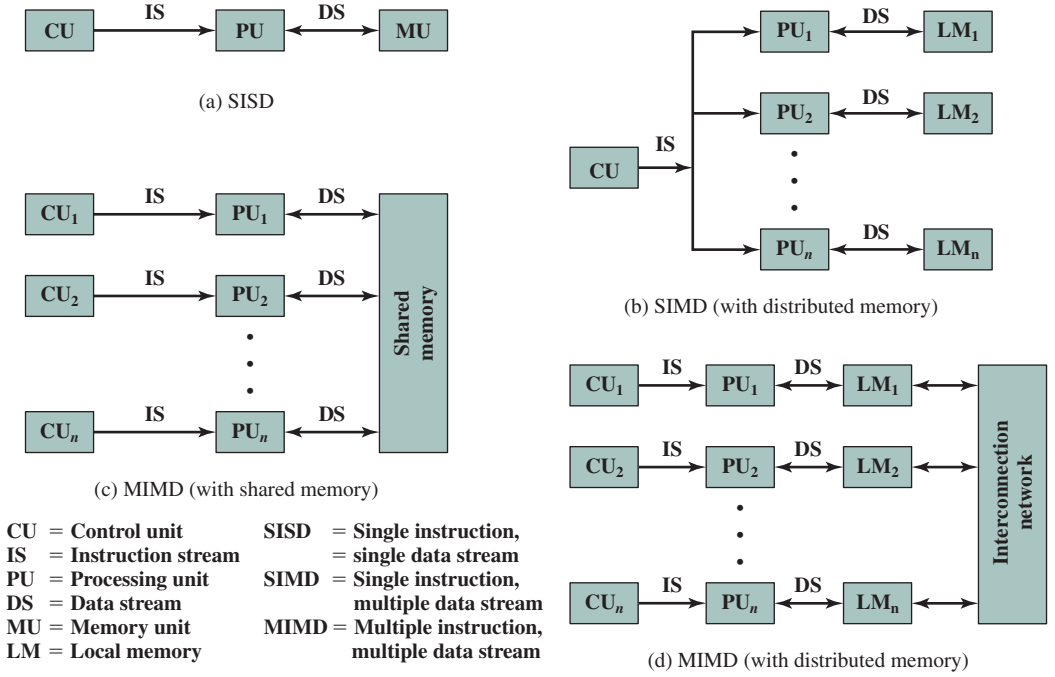


Figure 17.2 Alternative Computer Organizations

unit operates on a single data stream (DS) from a memory unit (MU). With an SIMD, there is still a single control unit, now feeding a single instruction stream to multiple PUs. Each PU may have its own dedicated memory (illustrated in Figure 17.2b), or there may be a shared memory. Finally, with the MIMD, there are multiple control units, each feeding a separate instruction stream to its own PU. The MIMD may be a shared-memory multiprocessor (Figure 17.2c) or a distributed-memory multicomputer (Figure 17.2d).

The design issues relating to SMPs, clusters, and NUMAs are complex, involving issues relating to physical organization, interconnection structures, interprocessor communication, operating system design, and application software techniques. Our concern here is primarily with organization, although we touch briefly on operating system design issues.

17.2 SYMMETRIC MULTIPROCESSORS

Until fairly recently, virtually all single-user personal computers and most workstations contained a single general-purpose microprocessor. As demands for performance increase and as the cost of microprocessors continues to drop, vendors have introduced systems with an SMP organization. The term *SMP* refers to a computer hardware architecture and also to the operating system behavior that reflects that architecture. An SMP can be defined as a standalone computer system with the following characteristics:

1. There are two or more similar processors of comparable capability.
2. These processors share the same main memory and I/O facilities and are interconnected by a bus or other internal connection scheme, such that memory access time is approximately the same for each processor.
3. All processors share access to I/O devices, either through the same channels or through different channels that provide paths to the same device.
4. All processors can perform the same functions (hence the term *symmetric*).
5. The system is controlled by an integrated operating system that provides interaction between processors and their programs at the job, task, file, and data element levels.

Points 1 to 4 should be self-explanatory. Point 5 illustrates one of the contrasts with a loosely coupled multiprocessing system, such as a cluster. In the latter, the physical unit of interaction is usually a message or complete file. In an SMP, individual data elements can constitute the level of interaction, and there can be a high degree of cooperation between processes.

The operating system of an SMP schedules processes or threads across all of the processors. An SMP organization has a number of potential advantages over a uniprocessor organization, including the following:

- **Performance:** If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type (Figure 17.3).

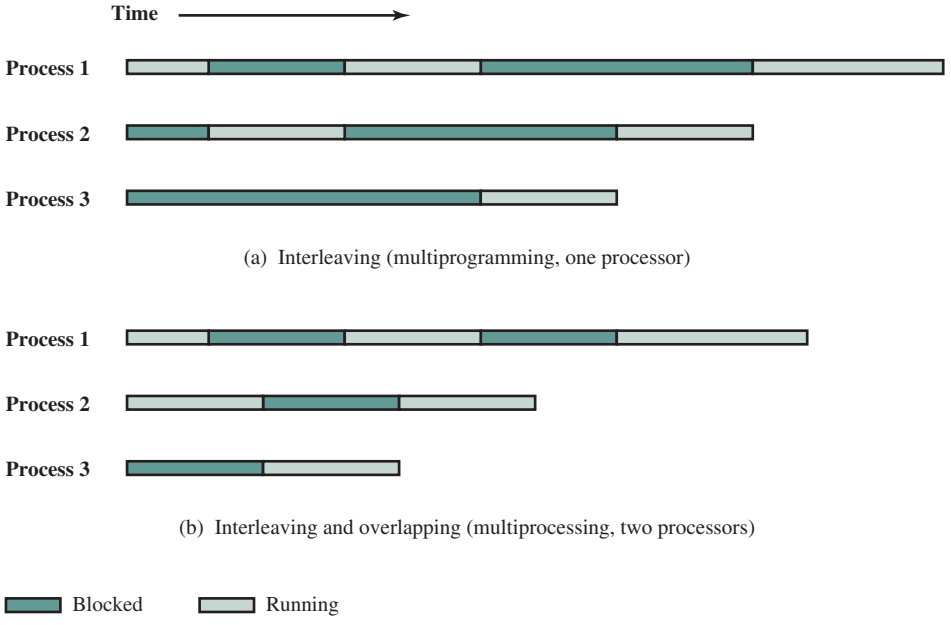


Figure 17.3 Multiprogramming and Multiprocessing

- **Availability:** In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the machine. Instead, the system can continue to function at reduced performance.
- **Incremental growth:** A user can enhance the performance of a system by adding an additional processor.
- **Scaling:** Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.

It is important to note that these are potential, rather than guaranteed, benefits. The operating system must provide tools and functions to exploit the parallelism in an SMP system.

An attractive feature of an SMP is that the existence of multiple processors is transparent to the user. The operating system takes care of scheduling of threads or processes on individual processors and of synchronization among processors.

Organization

Figure 17.4 depicts in general terms the organization of a multiprocessor system. There are two or more processors. Each processor is self-contained, including a control unit, ALU, registers, and, typically, one or more levels of cache. Each processor has access to a shared main memory and the I/O devices through some form of interconnection mechanism. The processors can communicate with each other through memory (messages and status information left in common data areas). It may also be possible for processors to exchange signals directly. The memory is often organized

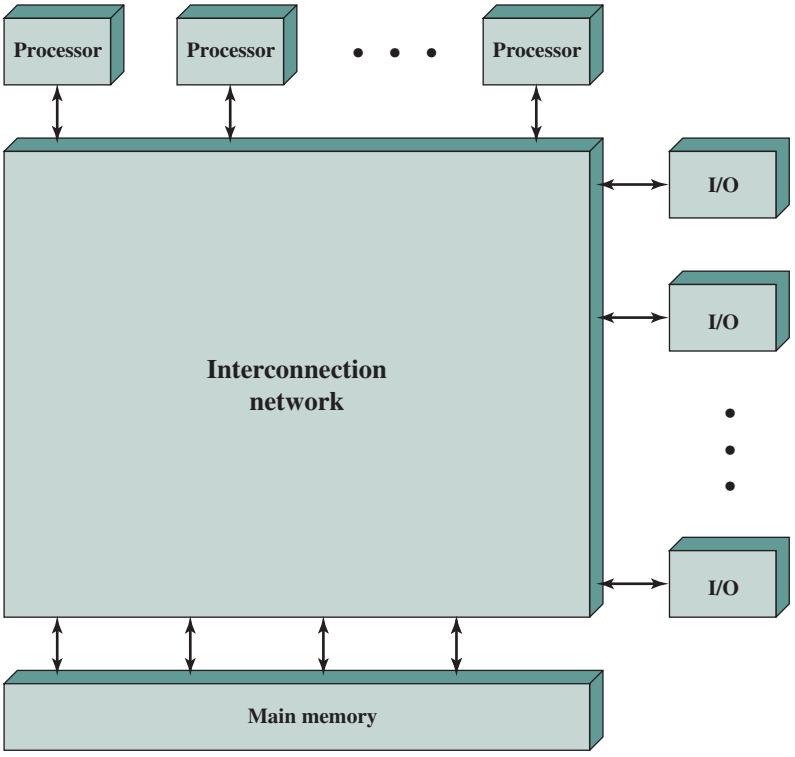


Figure 17.4 Generic Block Diagram of a Tightly Coupled Multiprocessor

so that multiple simultaneous accesses to separate blocks of memory are possible. In some configurations, each processor may also have its own private main memory and I/O channels in addition to the shared resources.

The most common organization for personal computers, workstations, and servers is the time-shared bus. The time-shared bus is the simplest mechanism for constructing a multiprocessor system (Figure 17.5). The structure and interfaces are basically the same as for a single-processor system that uses a bus interconnection. The bus consists of control, address, and data lines. To facilitate DMA transfers from I/O subsystems to processors, the following features are provided:

- **Addressing:** It must be possible to distinguish modules on the bus to determine the source and destination of data.
- **Arbitration:** Any I/O module can temporarily function as "master." A mechanism is provided to arbitrate competing requests for bus control, using some sort of priority scheme.
- **Time-sharing:** When one module is controlling the bus, other modules are locked out and must, if necessary, suspend operation until bus access is achieved.

These uniprocessor features are directly usable in an SMP organization. In this latter case, there are now multiple processors as well as multiple I/O processors all attempting to gain access to one or more memory modules via the bus.

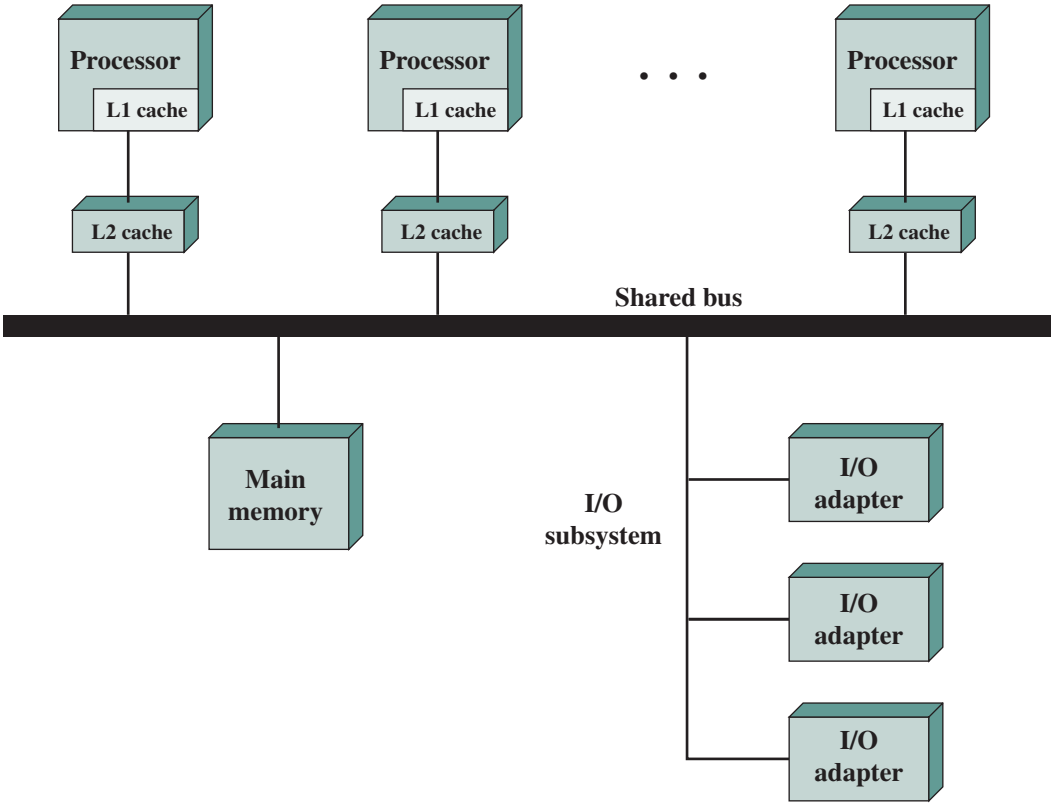


Figure 17.5 Symmetric Multiprocessor Organization

The bus organization has several attractive features:

- **Simplicity:** This is the simplest approach to multiprocessor organization. The physical interface and the addressing, arbitration, and time-sharing logic of each processor remain the same as in a single-processor system.
- **Flexibility:** It is generally easy to expand the system by attaching more processors to the bus.
- **Reliability:** The bus is essentially a passive medium, and the failure of any attached device should not cause failure of the whole system.

The main drawback to the bus organization is performance. All memory references pass through the common bus. Thus, the bus cycle time limits the speed of the system. To improve performance, it is desirable to equip each processor with a cache memory. This should reduce the number of bus accesses dramatically. Typically, workstation and PC SMPs have two levels of cache, with the L1 cache internal (same chip as the processor) and the L2 cache either internal or external. Some processors now employ a L3 cache as well.

The use of caches introduces some new design considerations. Because each local cache contains an image of a portion of memory, if a word is altered in one

cache, it could conceivably invalidate a word in another cache. To prevent this, the other processors must be alerted that an update has taken place. This problem is known as the *cache coherence* problem and is typically addressed in hardware rather than by the operating system. We address this issue in Section 17.4.

Multiprocessor Operating System Design Considerations

An SMP operating system manages processor and other computer resources so that the user perceives a single operating system controlling system resources. In fact, such a configuration should appear as a single-processor multiprogramming system. In both the SMP and uniprocessor cases, multiple jobs or processes may be active at one time, and it is the responsibility of the operating system to schedule their execution and to allocate resources. A user may construct applications that use multiple processes or multiple threads within processes without regard to whether a single processor or multiple processors will be available. Thus, a multiprocessor operating system must provide all the functionality of a multiprogramming system plus additional features to accommodate multiple processors. Among the key design issues:

- **Simultaneous concurrent processes:** OS routines need to be reentrant to allow several processors to execute the same IS code simultaneously. With multiple processors executing the same or different parts of the OS, OS tables and management structures must be managed properly to avoid deadlock or invalid operations.
- **Scheduling:** Any processor may perform scheduling, so conflicts must be avoided. The scheduler must assign ready processes to available processors.
- **Synchronization:** With multiple active processes having potential access to shared address spaces or shared I/O resources, care must be taken to provide effective synchronization. Synchronization is a facility that enforces mutual exclusion and event ordering.
- **Memory management:** Memory management on a multiprocessor must deal with all of the issues found on uniprocessor machines, as is discussed in Chapter 8. In addition, the operating system needs to exploit the available hardware parallelism, such as multiported memories, to achieve the best performance. The paging mechanisms on different processors must be coordinated to enforce consistency when several processors share a page or segment and to decide on page replacement.
- **Reliability and fault tolerance:** The operating system should provide graceful degradation in the face of processor failure. The scheduler and other portions of the operating system must recognize the loss of a processor and restructure management tables accordingly.

17.3 CACHE COHERENCE AND THE MESI PROTOCOL

In contemporary multiprocessor systems, it is customary to have one or two levels of cache associated with each processor. This organization is essential to achieve reasonable performance. It does, however, create a problem known as the *cache coherence*

problem. The essence of the problem is this: Multiple copies of the same data can exist in different caches simultaneously, and if processors are allowed to update their own copies freely, an inconsistent view of memory can result. In Chapter 4 we defined two common write policies:

- **Write back:** Write operations are usually made only to the cache. Main memory is only updated when the corresponding cache line is evicted from the cache.
- **Write through:** All write operations are made to main memory as well as to the cache, ensuring that main memory is always valid.

It is clear that a write-back policy can result in inconsistency. If two caches contain the same line, and the line is updated in one cache, the other cache will unknowingly have an invalid value. Subsequent reads to that invalid line produce invalid results. Even with the write-through policy, inconsistency can occur unless other caches monitor the memory traffic or receive some direct notification of the update.

In this section, we will briefly survey various approaches to the cache coherence problem and then focus on the approach that is most widely used: the MESI (modified/exclusive/shared/invalid) protocol. A version of this protocol is used on both the x86 architecture.

For any cache coherence protocol, the objective is to let recently used local variables get into the appropriate cache and stay there through numerous reads and write, while using the protocol to maintain consistency of shared variables that might be in multiple caches at the same time. Cache coherence approaches have generally been divided into software and hardware approaches. Some implementations adopt a strategy that involves both software and hardware elements. Nevertheless, the classification into software and hardware approaches is still instructive and is commonly used in surveying cache coherence strategies.

Software Solutions

Software cache coherence schemes attempt to avoid the need for additional hardware circuitry and logic by relying on the compiler and operating system to deal with the problem. Software approaches are attractive because the overhead of detecting potential problems is transferred from run time to compile time, and the design complexity is transferred from hardware to software. On the other hand, compile-time software approaches generally must make conservative decisions, leading to inefficient cache utilization.

Compiler-based coherence mechanisms perform an analysis on the code to determine which data items may become unsafe for caching, and they mark those items accordingly. The operating system or hardware then prevents noncacheable items from being cached.

The simplest approach is to prevent any shared data variables from being cached. This is too conservative, because a shared data structure may be exclusively used during some periods and may be effectively read-only during other periods. It is only during periods when at least one process may update the variable and at least one other process may access the variable that cache coherence is an issue.

More efficient approaches analyze the code to determine safe periods for shared variables. The compiler then inserts instructions into the generated code to enforce cache coherence during the critical periods. A number of techniques have been developed for performing the analysis and for enforcing the results; see [LILJ93] and [STEN90] for surveys.

Hardware Solutions

Hardware-based solutions are generally referred to as cache coherence protocols. These solutions provide dynamic recognition at run time of potential inconsistency conditions. Because the problem is only dealt with when it actually arises, there is more effective use of caches, leading to improved performance over a software approach. In addition, these approaches are transparent to the programmer and the compiler, reducing the software development burden.

Hardware schemes differ in a number of particulars, including where the state information about data lines is held, how that information is organized, where coherence is enforced, and the enforcement mechanisms. In general, hardware schemes can be divided into two categories: **directory protocols** and **snoopy protocols**.

DIRECTORY PROTOCOLS Directory protocols collect and maintain information about where copies of lines reside. Typically, there is a centralized controller that is part of the main memory controller, and a directory that is stored in main memory. The directory contains global state information about the contents of the various local caches. When an individual cache controller makes a request, the centralized controller checks and issues necessary commands for data transfer between memory and caches or between caches. It is also responsible for keeping the state information up to date; therefore, every local action that can affect the global state of a line must be reported to the central controller.

Typically, the controller maintains information about which processors have a copy of which lines. Before a processor can write to a local copy of a line, it must request exclusive access to the line from the controller. Before granting this exclusive access, the controller sends a message to all processors with a cached copy of this line, forcing each processor to invalidate its copy. After receiving acknowledgments back from each such processor, the controller grants exclusive access to the requesting processor. When another processor tries to read a line that is exclusively granted to another processor, it will send a miss notification to the controller. The controller then issues a command to the processor holding that line that requires the processor to do a write back to main memory. The line may now be shared for reading by the original processor and the requesting processor.

Directory schemes suffer from the drawbacks of a central bottleneck and the overhead of communication between the various cache controllers and the central controller. However, they are effective in large-scale systems that involve multiple buses or some other complex interconnection scheme.

SNOOPY PROTOCOLS Snoopy protocols distribute the responsibility for maintaining cache coherence among all of the cache controllers in a multiprocessor. A cache must recognize when a line that it holds is shared with other

caches. When an update action is performed on a shared cache line, it must be announced to all other caches by a broadcast mechanism. Each cache controller is able to "snoop" on the network to observe these broadcasted notifications, and react accordingly.

Snoopy protocols are ideally suited to a bus-based multiprocessor, because the shared bus provides a simple means for broadcasting and snooping. However, because one of the objectives of the use of local caches is to avoid bus accesses, care must be taken that the increased bus traffic required for broadcasting and snooping does not cancel out the gains from the use of local caches.

Two basic approaches to the snoopy protocol have been explored: write invalidate and write update (or write broadcast). With a write-invalidate protocol, there can be multiple readers but only one writer at a time. Initially, a line may be shared among several caches for reading purposes. When one of the caches wants to perform a write to the line, it first issues a notice that invalidates that line in the other caches, making the line exclusive to the writing cache. Once the line is exclusive, the owning processor can make cheap local writes until some other processor requires the same line.

With a write-update protocol, there can be multiple writers as well as multiple readers. When a processor wishes to update a shared line, the word to be updated is distributed to all others, and caches containing that line can update it.

Neither of these two approaches is superior to the other under all circumstances. Performance depends on the number of local caches and the pattern of memory reads and writes. Some systems implement adaptive protocols that employ both write-invalidate and write-update mechanisms.

The write-invalidate approach is the most widely used in commercial multiprocessor systems, such as the x86 architecture. It marks the state of every cache line (using two extra bits in the cache tag) as modified, exclusive, shared, or invalid. For this reason, the write-invalidate protocol is called MESI. In the remainder of this section, we will look at its use among local caches across a multiprocessor. For simplicity in the presentation, we do not examine the mechanisms involved in coordinating among both level 1 and level 2 locally as well as at the same time coordinating across the distributed multiprocessor. This would not add any new principles but would greatly complicate the discussion.

The MESI Protocol

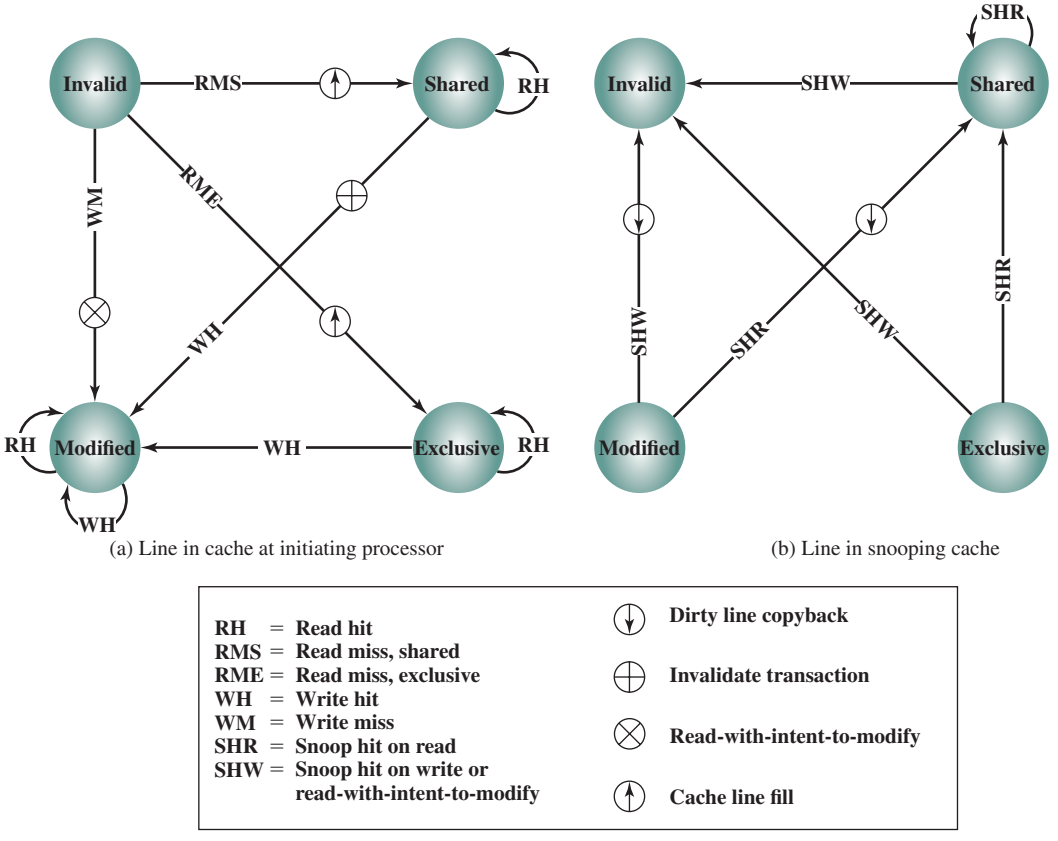
To provide cache consistency on an SMP, the data cache often supports a protocol known as MESI. For MESI, the data cache includes two status bits per tag, so that each line can be in one of four states:

- **Modified:** The line in the cache has been modified (different from main memory) and is available only in this cache.
- **Exclusive:** The line in the cache is the same as that in main memory and is not present in any other cache.
- **Shared:** The line in the cache is the same as that in main memory and may be present in another cache.
- **Invalid:** The line in the cache does not contain valid data.

Table 17.1 MESI Cache Line States

	M Modified	E Exclusive	S Shared	I Invalid
This cache line valid?	Yes	Yes	Yes	No
The memory copy is …	out of date	valid	valid	—
Copies exist in other caches?	No	No	Maybe	Maybe
A write to this line …	does not go to bus	does not go to bus	goes to bus and updates cache	goes directly to bus

Table 17.1 summarizes the meaning of the four states. Figure 17.6 displays a state diagram for the MESI protocol. Keep in mind that each line of the cache has its own state bits and therefore its own realization of the state diagram. Figure 17.6a shows the transitions that occur due to actions initiated by the processor attached to this cache. Figure 17.6b shows the transitions that occur due to events that are snooped on the common bus. This presentation of separate state diagrams for processor-initiated and bus-initiated actions helps to clarify the logic of the MESI



(a) Line in cache at initiating processor

(b) Line in snooping cache

Figure 17.6 MESI State Transition Diagram

protocol. At any time a cache line is in a single state. If the next event is from the attached processor, then the transition is dictated by Figure 17.6a and if the next event is from the bus, the transition is dictated by Figure 17.6b. Let us look at these transitions in more detail.

READ MISS When a read miss occurs in the local cache, the processor initiates a memory read to read the line of main memory containing the missing address. The processor inserts a signal on the bus that alerts all other processor/cache units to snoop the transaction. There are a number of possible outcomes:

- If one other cache has a clean (unmodified since read from memory) copy of the line in the exclusive state, it returns a signal indicating that it shares this line. The responding processor then transitions the state of its copy from exclusive to shared, and the initiating processor reads the line from main memory and transitions the line in its cache from invalid to shared.
- If one or more caches have a clean copy of the line in the shared state, each of them signals that it shares the line. The initiating processor reads the line and transitions the line in its cache from invalid to shared.
- If one other cache has a modified copy of the line, then that cache blocks the memory read and provides the line to the requesting cache over the shared bus. The responding cache then changes its line from modified to shared.[1] The line sent to the requesting cache is also received and processed by the memory controller, which stores the block in memory.
- If no other cache has a copy of the line (clean or modified), then no signals are returned. The initiating processor reads the line and transitions the line in its cache from invalid to exclusive.

READ HIT When a read hit occurs on a line currently in the local cache, the processor simply reads the required item. There is no state change: The state remains modified, shared, or exclusive.

WRITE MISS When a write miss occurs in the local cache, the processor initiates a memory read to read the line of main memory containing the missing address. For this purpose, the processor issues a signal on the bus that means *read-with-intent-to-modify* (RWITM). When the line is loaded, it is immediately marked modified. With respect to other caches, two possible scenarios precede the loading of the line of data.

First, some other cache may have a modified copy of this line (state = modify). In this case, the alerted processor signals the initiating processor that another processor has a modified copy of the line. The initiating processor surrenders the bus and waits. The other processor gains access to the bus, writes the modified cache

[1] In some implementations, the cache with the modified line signals the initiating processor to retry. Meanwhile, the processor with the modified copy seizes the bus, writes the modified line back to main memory, and transitions the line in its cache from modified to shared. Subsequently, the requesting processor tries again and finds that one or more processors have a clean copy of the line in the shared state, as described in the preceding point.

line back to main memory, and transitions the state of the cache line to invalid (because the initiating processor is going to modify this line). Subsequently, the initiating processor will again issue a signal to the bus of RWITM and then read the line from main memory, modify the line in the cache, and mark the line in the modified state.

The second scenario is that no other cache has a modified copy of the requested line. In this case, no signal is returned, and the initiating processor proceeds to read in the line and modify it. Meanwhile, if one or more caches have a clean copy of the line in the shared state, each cache invalidates its copy of the line, and if one cache has a clean copy of the line in the exclusive state, it invalidates its copy of the line.

WRITE HIT When a write hit occurs on a line currently in the local cache, the effect depends on the current state of that line in the local cache:

- **Shared:** Before performing the update, the processor must gain exclusive ownership of the line. The processor signals its intent on the bus. Each processor that has a shared copy of the line in its cache transitions the sector from shared to invalid. The initiating processor then performs the update and transitions its copy of the line from shared to modified.
- **Exclusive:** The processor already has exclusive control of this line, and so it simply performs the update and transitions its copy of the line from exclusive to modified.
- **Modified:** The processor already has exclusive control of this line and has the line marked as modified, and so it simply performs the update.

L1–L2 CACHE CONSISTENCY We have so far described cache coherency protocols in terms of the cooperate activity among caches connected to the same bus or other SMP interconnection facility. Typically, these caches are L2 caches, and each processor also has an L1 cache that does not connect directly to the bus and that therefore cannot engage in a snoopy protocol. Thus, some scheme is needed to maintain data integrity across both levels of cache and across all caches in the SMP configuration.

The strategy is to extend the MESI protocol (or any cache coherence protocol) to the L1 caches. Thus, each line in the L1 cache includes bits to indicate the state. In essence, the objective is the following: for any line that is present in both an L2 cache and its corresponding L1 cache, the L1 line state should track the state of the L2 line. A simple means of doing this is to adopt the write-through policy in the L1 cache; in this case the write through is to the L2 cache and not to the memory. The L1 write-through policy forces any modification to an L1 line out to the L2 cache and therefore makes it visible to other L2 caches. The use of the L1 write-through policy requires that the L1 content must be a subset of the L2 content. This in turn suggests that the associativity of the L2 cache should be equal to or greater than that of the L1 associativity. The L1 write-through policy is used in the IBM S/390 SMP.

If the L1 cache has a write-back policy, the relationship between the two caches is more complex. There are several approaches to maintaining, a topic beyond our scope.

17.4 MULTITHREADING AND CHIP MULTIPROCESSORS

The most important measure of performance for a processor is the rate at which it executes instructions. This can be expressed as

$$\text{MIPS rate} = f \times IPC$$

where f is the processor clock frequency, in MHz, and IPC (instructions per cycle) is the average number of instructions executed per cycle. Accordingly, designers have pursued the goal of increased performance on two fronts: increasing clock frequency and increasing the number of instructions executed or, more properly, the number of instructions that complete during a processor cycle. As we have seen in earlier chapters, designers have increased IPC by using an instruction pipeline and then by using multiple parallel instruction pipelines in a superscalar architecture. With pipelined and multiple-pipeline designs, the principal problem is to maximize the utilization of each pipeline stage. To improve throughput, designers have created ever more complex mechanisms, such as executing some instructions in a different order from the way they occur in the instruction stream and beginning execution of instructions that may never be needed. But as was discussed in Section 2.2, this approach may be reaching a limit due to complexity and power consumption concerns.

An alternative approach, which allows for a high degree of instruction-level parallelism without increasing circuit complexity or power consumption, is called multithreading. In essence, the instruction stream is divided into several smaller streams, known as threads, such that the threads can be executed in parallel.

The variety of specific multithreading designs, realized in both commercial systems and experimental systems, is vast. In this section, we give a brief survey of the major concepts.

Implicit and Explicit Multithreading

The concept of thread used in discussing multithreaded processors may or may not be the same as the concept of software threads in a multiprogrammed operating system. It will be useful to define terms briefly:

- **Process:** An instance of a program running on a computer. A process embodies two key characteristics:
  - **Resource ownership:** A process includes a virtual address space to hold the process image; the process image is the collection of program, data, stack, and attributes that define the process. From time to time, a process may be allocated control or ownership of resources, such as main memory, I/O channels, I/O devices, and files.
  - **Scheduling/execution:** The execution of a process follows an execution path (trace) through one or more programs. This execution may be interleaved with that of other processes. Thus, a process has an execution state (Running, Ready, etc.) and a dispatching priority and is the entity that is scheduled and dispatched by the operating system.

- **Process switch:** An operation that switches the processor from one process to another, by saving all the process control data, registers, and other information for the first and replacing them with the process information for the second.[2]
- **Thread:** A dispatchable unit of work within a process. It includes a processor context (which includes the program counter and stack pointer) and its own data area for a stack (to enable subroutine branching). A thread executes sequentially and is interruptible so that the processor can turn to another thread.
- **Thread switch:** The act of switching processor control from one thread to another within the same process. Typically, this type of switch is much less costly than a process switch.

Thus, a thread is concerned with scheduling and execution, whereas a process is concerned with both scheduling/execution and resource ownership. The multiple threads within a process share the same resources. This is why a thread switch is much less time consuming than a process switch. Traditional operating systems, such as earlier versions of unix, did not support threads. Most modern operating systems, such as Linux, other versions of unix, and Windows, do support thread. A distinction is made between user-level threads, which are visible to the application program, and kernel-level threads, which are visible only to the operating system. Both of these may be referred to as explicit threads, defined in software.

All of the commercial processors and most of the experimental processors so far have used explicit multithreading. These systems concurrently execute instructions from different explicit threads, either by interleaving instructions from different threads on shared pipelines or by parallel execution on parallel pipelines. Implicit multithreading refers to the concurrent execution of multiple threads extracted from a single sequential program. These implicit threads may be defined either statically by the compiler or dynamically by the hardware. In the remainder of this section we consider explicit multithreading.

Approaches to Explicit Multithreading

At minimum, a multithreaded processor must provide a separate program counter for each thread of execution to be executed concurrently. The designs differ in the amount and type of additional hardware used to support concurrent thread execution. In general, instruction fetching takes place on a thread basis. The processor treats each thread separately and may use a number of techniques for optimizing single-thread execution, including branch prediction, register renaming, and superscalar techniques. What is achieved is thread-level parallelism, which may provide for greatly improved performance when married to instruction-level parallelism.

Broadly speaking, there are four principal approaches to multithreading:

- **Interleaved multithreading:** This is also known as **fine-grained multithreading**. The processor deals with two or more thread contexts at a time, switching from one thread to another at each clock cycle. If a thread is blocked because

[2]The term *context switch* is often found in OS literature and textbooks. Unfortunately, although most of the literature uses this term to mean what is here called a process switch, other sources use it to mean a thread switch. To avoid ambiguity, the term is not used in this book.

of data dependencies or memory latencies, that thread is skipped and a ready thread is executed.

- **Blocked multithreading:** This is also known as **coarse-grained multithreading**. The instructions of a thread are executed successively until an event occurs that may cause delay, such as a cache miss. This event induces a switch to another thread. This approach is effective on an in-order processor that would stall the pipeline for a delay event such as a cache miss.
- **Simultaneous multithreading (SMT):** Instructions are simultaneously issued from multiple threads to the execution units of a superscalar processor. This combines the wide superscalar instruction issue capability with the use of multiple thread contexts.
- **Chip multiprocessing:** In this case, multiple cores are implemented on a single chip and each core handles separate threads. The advantage of this approach is that the available logic area on a chip is used effectively without depending on ever-increasing complexity in pipeline design. This is referred to as multicore; we examine this topic separately in Chapter 18.

For the first two approaches, instructions from different threads are not executed simultaneously. Instead, the processor is able to rapidly switch from one thread to another, using a different set of registers and other context information. This results in a better utilization of the processor's execution resources and avoids a large penalty due to cache misses and other latency events. The SMT approach involves true simultaneous execution of instructions from different threads, using replicated execution resources. Chip multiprocessing also enables simultaneous execution of instructions from different threads.

Figure 17.7, based on one in [UNGE02], illustrates some of the possible pipeline architectures that involve multithreading and contrasts these with approaches that do not use multithreading. Each horizontal row represents the potential issue slot or slots for a single execution cycle; that is, the width of each row corresponds to the maximum number of instructions that can be issued in a single clock cycle.[3] The vertical dimension represents the time sequence of clock cycles. An empty (shaded) slot represents an unused execution slot in one pipeline. A no-op is indicated by N.

The first three illustrations in Figure 17.7 show different approaches with a scalar (i.e., single-issue) processor:

- **Single-threaded scalar:** This is the simple pipeline found in traditional RISC and CISC machines, with no multithreading.
- **Interleaved multithreaded scalar:** This is the easiest multithreading approach to implement. By switching from one thread to another at each clock cycle, the pipeline stages can be kept fully occupied, or close to fully occupied. The hardware must be capable of switching from one thread context to another between cycles.

[3]Issue slots are the position from which instructions can be issued in a given clock cycle. Recall from Chapter 16 that instruction issue is the process of initiating instruction execution in the processor's functional units. This occurs when an instruction moves from the decode stage of the pipeline to the first execute stage of the pipeline.

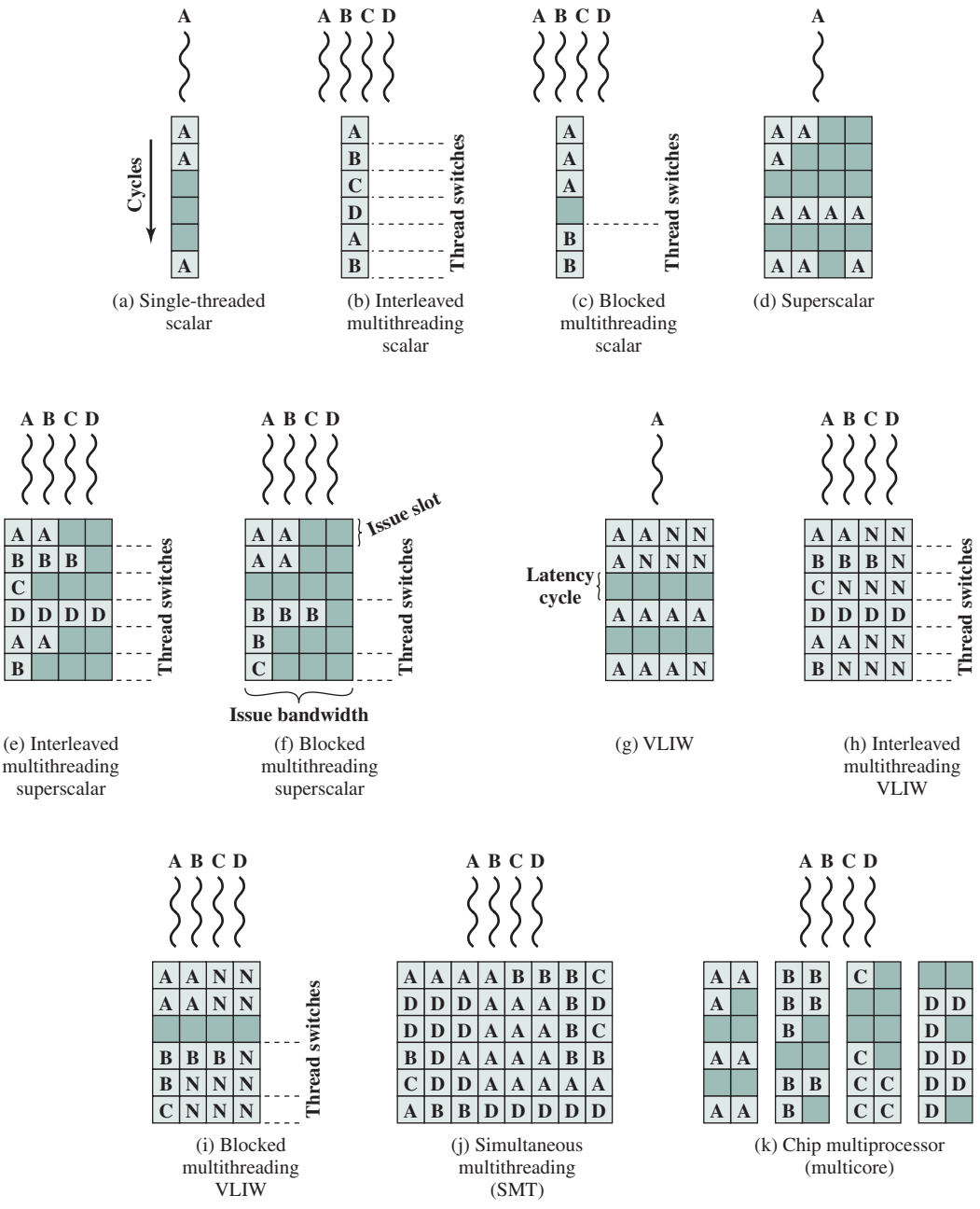


Figure 17.7 Approaches to Executing Multiple Threads

- **Blocked multithreaded scalar:** In this case, a single thread is executed until a latency event occurs that would stop the pipeline, at which time the processor switches to another thread.

Figure 17.7c shows a situation in which the time to perform a thread switch is one cycle, whereas Figure 17.7b shows that thread switching occurs in zero cycles.

In the case of interleaved multithreading, it is assumed that there are no control or data dependencies between threads, which simplifies the pipeline design and therefore should allow a thread switch with no delay. However, depending on the specific design and implementation, block multithreading may require a clock cycle to perform a thread switch, as illustrated in Figure 17.7. This is true if a fetched instruction triggers the thread switch and must be discarded from the pipeline [UNGE03].

Although interleaved multithreading appears to offer better processor utilization than blocked multithreading, it does so at the sacrifice of single-thread performance. The multiple threads compete for cache resources, which raises the probability of a cache miss for a given thread.

More opportunities for parallel execution are available if the processor can issue multiple instructions per cycle. Figures 17.7d through 17.7i illustrate a number of variations among processors that have hardware for issuing four instructions per cycle. In all these cases, only instructions from a single thread are issued in a single cycle. The following alternatives are illustrated:

- **Superscalar:** This is the basic superscalar approach with no multithreading. Until relatively recently, this was the most powerful approach to providing parallelism within a processor. Note that during some cycles, not all of the available issue slots are used. During these cycles, less than the maximum number of instructions is issued; this is referred to as *horizontal loss*. During other instruction cycles, no issue slots are used; these are cycles when no instructions can be issued; this is referred to as *vertical loss*.
- **Interleaved multithreading superscalar:** During each cycle, as many instructions as possible are issued from a single thread. With this technique, potential delays due to thread switches are eliminated, as previously discussed. However, the number of instructions issued in any given cycle is still limited by dependencies that exist within any given thread.
- **Blocked multithreaded superscalar:** Again, instructions from only one thread may be issued during any cycle, and blocked multithreading is used.
- **Very long instruction word (VLIW):** A VLIW architecture, such as IA-64, places multiple instructions in a single word. Typically, a VLIW is constructed by the compiler, which places operations that may be executed in parallel in the same word. In a simple VLIW machine (Figure 17.7g), if it is not possible to completely fill the word with instructions to be issued in parallel, no-ops are used.
- **Interleaved multithreading VLIW:** This approach should provide similar efficiencies to those provided by interleaved multithreading on a superscalar architecture.
- **Blocked multithreaded VLIW:** This approach should provide similar efficiencies to those provided by blocked multithreading on a superscalar architecture.

The final two approaches illustrated in Figure 17.7 enable the parallel, simultaneous execution of multiple threads:

- **Simultaneous multithreading:** Figure 17.7j shows a system capable of issuing 8 instructions at a time. If one thread has a high degree of instruction-level parallelism, it may on some cycles be able fill all of the horizontal slots. On other cycles, instructions from two or more threads may be issued. If sufficient

threads are active, it should usually be possible to issue the maximum number of instructions on each cycle, providing a high level of efficiency.

- **Chip multiprocessor (multicore):** Figure 17.7k shows a chip containing four cores, each of which has a two-issue superscalar processor. Each core is assigned a thread, from which it can issue up to two instructions per cycle. We discuss multicore computers in Chapter 18.

Comparing Figures 17.7j and 17.7k, we see that a chip multiprocessor with the same instruction issue capability as an SMT cannot achieve the same degree of instruction-level parallelism. This is because the chip multiprocessor is not able to hide latencies by issuing instructions from other threads. On the other hand, the chip multiprocessor should outperform a superscalar processor with the same instruction issue capability, because the horizontal losses will be greater for the superscalar processor. In addition, it is possible to use multithreading within each of the cores on a chip multiprocessor, and this is done on some contemporary machines.

17.5 CLUSTERS

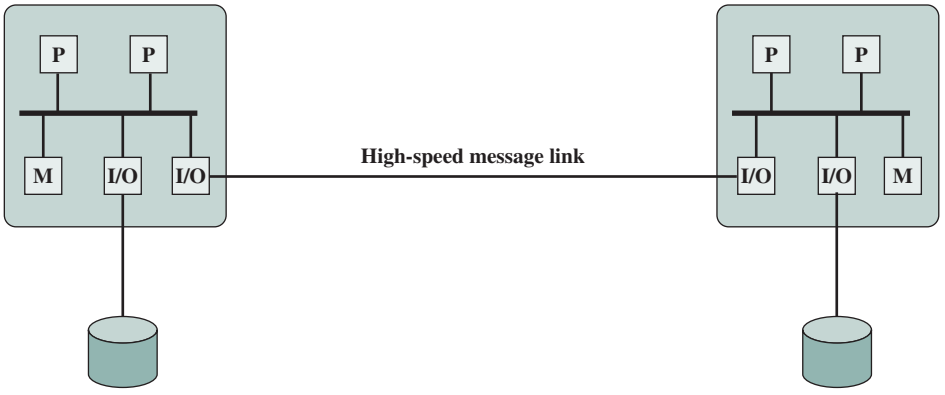
An important and relatively recent development computer system design is clustering. Clustering is an alternative to symmetric multiprocessing as an approach to providing high performance and high availability and is particularly attractive for server applications. We can define a cluster as a group of interconnected, whole computers working together as a unified computing resource that can create the illusion of being one machine. The term *whole computer* means a system that can run on its own, apart from the cluster; in the literature, each computer in a cluster is typically referred to as a *node*.

[BREW97] lists four benefits that can be achieved with clustering. These can also be thought of as objectives or design requirements:

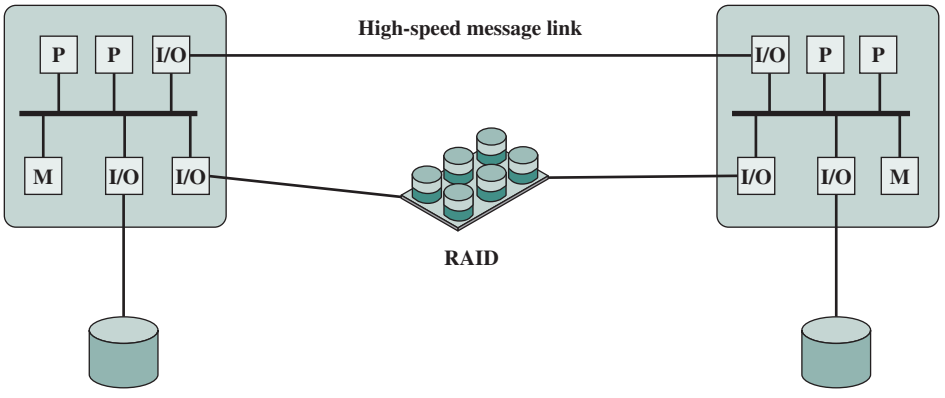
- **Absolute scalability:** It is possible to create large clusters that far surpass the power of even the largest standalone machines. A cluster can have tens, hundreds, or even thousands of machines, each of which is a multiprocessor.
- **Incremental scalability:** A cluster is configured in such a way that it is possible to add new systems to the cluster in small increments. Thus, a user can start out with a modest system and expand it as needs grow, without having to go through a major upgrade in which an existing small system is replaced with a larger system.
- **High availability:** Because each node in a cluster is a standalone computer, the failure of one node does not mean loss of service. In many products, fault tolerance is handled automatically in software.
- **Superior price/performance:** By using commodity building blocks, it is possible to put together a cluster with equal or greater computing power than a single large machine, at much lower cost.

Cluster Configurations

In the literature, clusters are classified in a number of different ways. Perhaps the simplest classification is based on whether the computers in a cluster share access to the same disks. Figure 17.8a shows a two-node cluster in which the only interconnection



(a) Standby server with no shared disk



(b) Shared Disk

Figure 17.8 Cluster Configurations

is by means of a high-speed link that can be used for message exchange to coordinate cluster activity. The link can be a LAN that is shared with other computers that are not part of the cluster or the link can be a dedicated interconnection facility. In the latter case, one or more of the computers in the cluster will have a link to a LAN or WAN so that there is a connection between the server cluster and remote client systems. Note that in the figure, each computer is depicted as being a multiprocessor. This is not necessary but does enhance both performance and availability.

In the simple classification depicted in Figure 17.8, the other alternative is a shared-disk cluster. In this case, there generally is still a message link between nodes. In addition, there is a disk subsystem that is directly linked to multiple computers within the cluster. In this figure, the common disk subsystem is a RAID system. The use of RAID or some similar redundant disk technology is common in clusters so that the high availability achieved by the presence of multiple computers is not compromised by a shared disk that is a single point of failure.

A clearer picture of the range of cluster options can be gained by looking at functional alternatives. Table 17.2 provides a useful classification along functional lines, which we now discuss.

Table 17.2 Clustering Methods: Benefits and Limitations

Clustering Method	Description	Benefits	Limitations
Passive Standby	A secondary server takes over in case of primary server failure.	Easy to implement.	High cost because the secondary server is unavailable for other processing tasks.
Active Secondary:	The secondary server is also used for processing tasks.	Reduced cost because secondary servers can be used for processing.	Increased complexity.
Separate Servers	Separate servers have their own disks. Data is continuously copied from primary to secondary server.	High availability.	High network and server overhead due to copying operations.
Servers Connected to Disks	Servers are cabled to the same disks, but each server owns its disks. If one server fails, its disks are taken over by the other server.	Reduced network and server overhead due to elimination of copying operations.	Usually requires disk mirroring or RAID technology to compensate for risk of disk failure.
Servers Share Disks	Multiple servers simultaneously share access to disks.	Low network and server overhead. Reduced risk of downtime caused by disk failure.	Requires lock manager software. Usually used with disk mirroring or RAID technology.

A common, older method, known as **passive standby**, is simply to have one computer handle all of the processing load while the other computer remains inactive, standing by to take over in the event of a failure of the primary. To coordinate the machines, the active, or primary, system periodically sends a "heartbeat" message to the standby machine. Should these messages stop arriving, the standby assumes that the primary server has failed and puts itself into operation. This approach increases availability but does not improve performance. Further, if the only information that is exchanged between the two systems is a heartbeat message, and if the two systems do not share common disks, then the standby provides a functional backup but has no access to the databases managed by the primary.

The passive standby is generally not referred to as a cluster. The term *cluster* is reserved for multiple interconnected computers that are all actively doing processing while maintaining the image of a single system to the outside world. The term **active secondary** is often used in referring to this configuration. Three classifications of clustering can be identified: separate servers, shared nothing, and shared memory.

In one approach to clustering, each computer is a **separate server** with its own disks and there are no disks shared between systems (Figure 17.8a). This arrangement provides high performance as well as high availability. In this case, some type of management or scheduling software is needed to assign incoming client requests to servers so that the load is balanced and high utilization is achieved. It is desirable to have a failover capability, which means that if a computer fails while executing an application, another computer in the cluster can pick up and complete

the application. For this to happen, data must constantly be copied among systems so that each system has access to the current data of the other systems. The overhead of this data exchange ensures high availability at the cost of a performance penalty.

To reduce the communications overhead, most clusters now consist of servers connected to common disks (Figure 17.8b). In one variation on this approach, called **shared nothing**, the common disks are partitioned into volumes, and each volume is owned by a single computer. If that computer fails, the cluster must be reconfigured so that some other computer has ownership of the volumes of the failed computer.

It is also possible to have multiple computers share the same disks at the same time (called the **shared disk** approach), so that each computer has access to all of the volumes on all of the disks. This approach requires the use of some type of locking facility to ensure that data can only be accessed by one computer at a time.

Operating System Design Issues

Full exploitation of a cluster hardware configuration requires some enhancements to a single-system operating system.

FAILURE MANAGEMENT How failures are managed by a cluster depends on the clustering method used (Table 17.2). In general, two approaches can be taken to dealing with failures: highly available clusters and fault-tolerant clusters. A highly available cluster offers a high probability that all resources will be in service. If a failure occurs, such as a system goes down or a disk volume is lost, then the queries in progress are lost. Any lost query, if retried, will be serviced by a different computer in the cluster. However, the cluster operating system makes no guarantee about the state of partially executed transactions. This would need to be handled at the application level.

A fault-tolerant cluster ensures that all resources are always available. This is achieved by the use of redundant shared disks and mechanisms for backing out uncommitted transactions and committing completed transactions.

The function of switching applications and data resources over from a failed system to an alternative system in the cluster is referred to as **failover**. A related function is the restoration of applications and data resources to the original system once it has been fixed; this is referred to as **failback**. Failback can be automated, but this is desirable only if the problem is truly fixed and unlikely to recur. If not, automatic failback can cause subsequently failed resources to bounce back and forth between computers, resulting in performance and recovery problems.

LOAD BALANCING A cluster requires an effective capability for balancing the load among available computers. This includes the requirement that the cluster be incrementally scalable. When a new computer is added to the cluster, the load-balancing facility should automatically include this computer in scheduling applications. Middleware mechanisms need to recognize that services can appear on different members of the cluster and may migrate from one member to another.

PARALLELIZING COMPUTATION In some cases, effective use of a cluster requires executing software from a single application in parallel. [KAPP00] lists three general approaches to the problem:

- **Parallelizing compiler:** A parallelizing compiler determines, at compile time, which parts of an application can be executed in parallel. These are then split off to be assigned to different computers in the cluster. Performance depends on the nature of the problem and how well the compiler is designed. In general, such compilers are difficult to develop.
- **Parallelized application:** In this approach, the programmer writes the application from the outset to run on a cluster, and uses message passing to move data, as required, between cluster nodes. This places a high burden on the programmer but may be the best approach for exploiting clusters for some applications.
- **Parametric computing:** This approach can be used if the essence of the application is an algorithm or program that must be executed a large number of times, each time with a different set of starting conditions or parameters. A good example is a simulation model, which will run a large number of different scenarios and then develop statistical summaries of the results. For this approach to be effective, parametric processing tools are needed to organize, run, and manage the jobs in an effective manner.

Cluster Computer Architecture

Figure 17.9 shows a typical cluster architecture. The individual computers are connected by some high-speed LAN or switch hardware. Each computer is capable of operating independently. In addition, a middleware layer of software is installed in each computer to enable cluster operation. The cluster middleware provides a unified system image to the user, known as a single-system image. The middleware is also responsible for providing high availability, by means of load balancing and

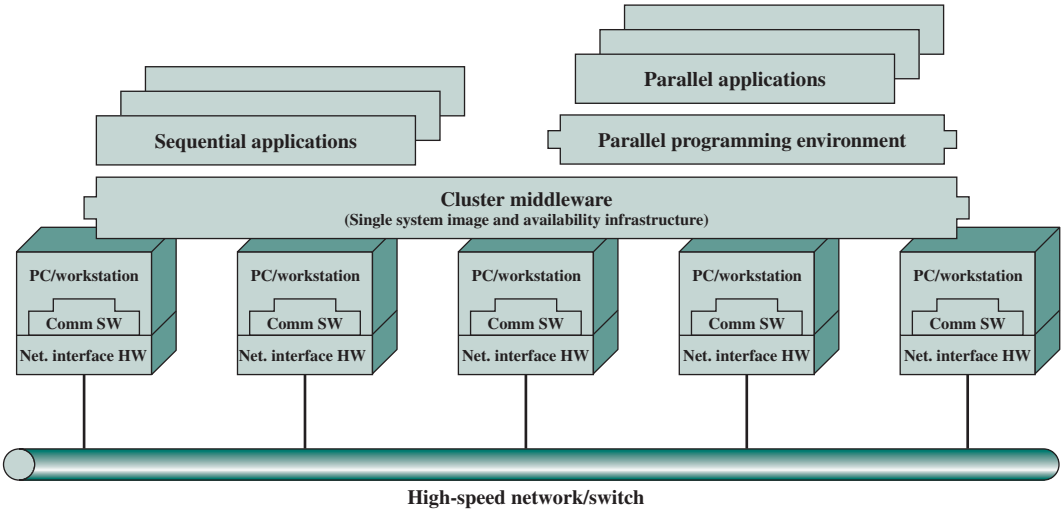


Figure 17.9 Cluster Computer Architecture [BUYY99]

responding to failures in individual components. [HWAN99] lists the following as desirable cluster middleware services and functions:

- **Single entry point:** A user logs onto the cluster rather than to an individual computer.
- **Single file hierarchy:** The user sees a single hierarchy of file directories under the same root directory.
- **Single control point:** There is a default workstation used for cluster management and control.
- **Single virtual networking:** Any node can access any other point in the cluster, even though the actual cluster configuration may consist of multiple interconnected networks. There is a single virtual network operation.
- **Single memory space:** Distributed shared memory enables programs to share variables.
- **Single job-management system:** Under a cluster job scheduler, a user can submit a job without specifying the host computer to execute the job.
- **Single user interface:** A common graphic interface supports all users, regardless of the workstation from which they enter the cluster.
- **Single I/O space:** Any node can remotely access any I/O peripheral or disk device without knowledge of its physical location.
- **Single process space:** A uniform process-identification scheme is used. A process on any node can create or communicate with any other process on a remote node.
- **Checkpointing:** This function periodically saves the process state and intermediate computing results, to allow rollback recovery after a failure.
- **Process migration:** This function enables load balancing.

The last four items on the preceding list enhance the availability of the cluster. The remaining items are concerned with providing a single system image.

Returning to Figure 17.9, a cluster will also include software tools for enabling the efficient execution of programs that are capable of parallel execution.

Blade Servers

A common implementation of the cluster approach is the blade server. A blade server is a server architecture that houses multiple server modules ("blades") in a single chassis. It is widely used in data centers to save space and improve system management. Either self-standing or rack mounted, the chassis provides the power supply, and each blade has its own processor, memory, and hard disk.

An example of the application is shown in Figure 17.10. The trend at large data centers, with substantial banks of blade servers, is the deployment of 10-Gbps ports on individual servers to handle the massive multimedia traffic provided by these servers. Such arrangements are stressing the on-site Ethernet switches needed to interconnect large numbers of servers. A 100-Gbps rate provides the bandwidth required to handle the increased traffic load. The 100-Gbps

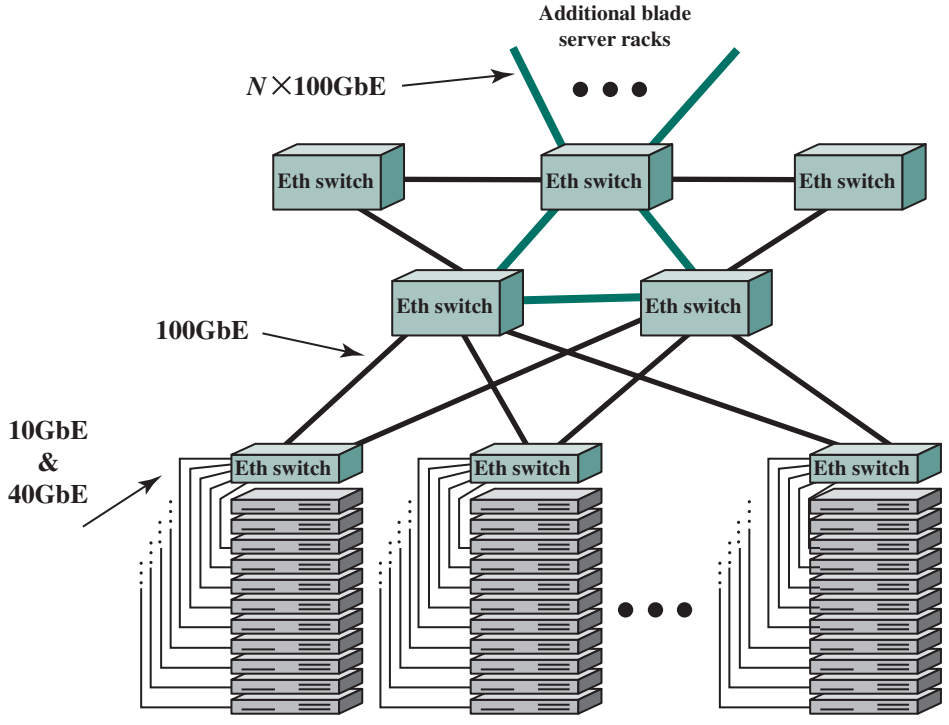


Figure 17.10 Example 100-Gbps Ethernet Configuration for Massive Blade Server Site

Ethernet switches are deployed in switch uplinks inside the data center as well as providing interbuilding, intercampus, wide area connections for enterprise networks.

Clusters Compared to SMP

Both clusters and symmetric multiprocessors provide a configuration with multiple processors to support high-demand applications. Both solutions are commercially available, although SMP schemes have been around far longer.

The main strength of the SMP approach is that an SMP is easier to manage and configure than a cluster. The SMP is much closer to the original single-processor model for which nearly all applications are written. The principal change required in going from a uniprocessor to an SMP is to the scheduler function. Another benefit of the SMP is that it usually takes up less physical space and draws less power than a comparable cluster. A final important benefit is that the SMP products are well established and stable.

Over the long run, however, the advantages of the cluster approach are likely to result in clusters dominating the high-performance server market. Clusters are far superior to SMPs in terms of incremental and absolute scalability. Clusters are also superior in terms of availability, because all components of the system can readily be made highly redundant.

17.6 NONUNIFORM MEMORY ACCESS

In terms of commercial products, the two common approaches to providing a multiple-processor system to support applications are SMPs and clusters. For some years, another approach, known as nonuniform memory access (NUMA), has been the subject of research and commercial NUMA products are now available.

Before proceeding, we should define some terms often found in the NUMA literature.

- **Uniform memory access (UMA):** All processors have access to all parts of main memory using loads and stores. The memory access time of a processor to all regions of memory is the same. The access times experienced by different processors are the same. The SMP organization discussed in Sections 17.2 and 17.3 is UMA.
- **Nonuniform memory access (NUMA):** All processors have access to all parts of main memory using loads and stores. The memory access time of a processor differs depending on which region of main memory is accessed. The last statement is true for all processors; however, for different processors, which memory regions are slower and which are faster differ.
- **Cache-coherent NUMA (CC-NUMA):** A NUMA system in which cache coherence is maintained among the caches of the various processors.

A NUMA system without cache coherence is more or less equivalent to a cluster. The commercial products that have received much attention recently are CC-NUMA systems, which are quite distinct from both SMPs and clusters. Usually, but unfortunately not always, such systems are in fact referred to in the commercial literature as CC-NUMA systems. This section is concerned only with CC-NUMA systems.

Motivation

With an SMP system, there is a practical limit to the number of processors that can be used. An effective cache scheme reduces the bus traffic between any one processor and main memory. As the number of processors increases, this bus traffic also increases. Also, the bus is used to exchange cache-coherence signals, further adding to the burden. At some point, the bus becomes a performance bottleneck. Performance degradation seems to limit the number of processors in an SMP configuration to somewhere between 16 and 64 processors. For example, Silicon Graphics' Power Challenge SMP is limited to 64 R10000 processors in a single system; beyond this number performance degrades substantially.

The processor limit in an SMP is one of the driving motivations behind the development of cluster systems. However, with a cluster, each node has its own private main memory; applications do not see a large global memory. In effect, coherency is maintained in software rather than hardware. This memory granularity affects performance and, to achieve maximum performance, software must be tailored to this environment. One approach to achieving large-scale multiprocessing while retaining the flavor of SMP is NUMA.

The objective with NUMA is to maintain a transparent system wide memory while permitting multiple multiprocessor nodes, each with its own bus or other internal interconnect system.

Organization

Figure 17.11 depicts a typical CC-NUMA organization. There are multiple independent nodes, each of which is, in effect, an SMP organization. Thus, each node contains multiple processors, each with its own L1 and L2 caches, plus main memory. The node is the basic building block of the overall CC-NUMA organization. For example, each Silicon Graphics Origin node includes two MIPS R10000 processors;

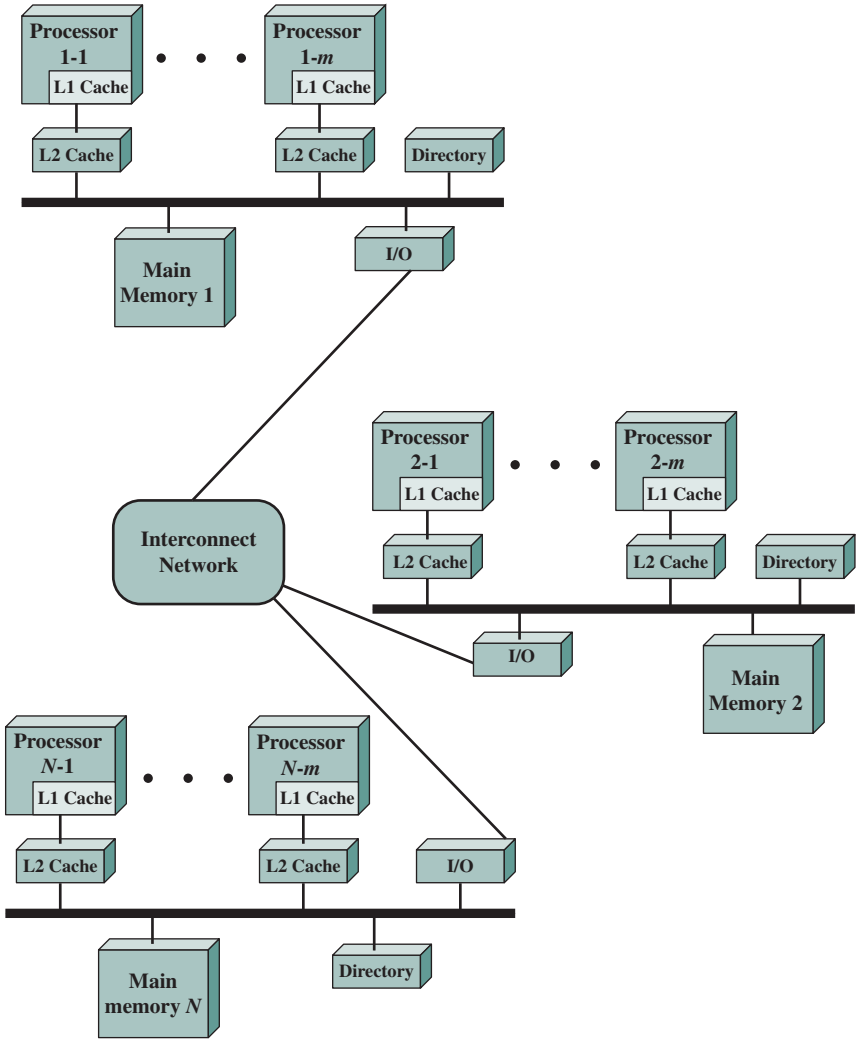


Figure 17.11 CC-NUMA Organization

each Sequent NUMA-Q node includes four Pentium II processors. The nodes are interconnected by means of some communications facility, which could be a switching mechanism, a ring, or some other networking facility.

Each node in the CC-NUMA system includes some main memory. From the point of view of the processors, however, there is only a single addressable memory, with each location having a unique system wide address. When a processor initiates a memory access, if the requested memory location is not in that processor's cache, then the L2 cache initiates a fetch operation. If the desired line is in the local portion of the main memory, the line is fetched across the local bus. If the desired line is in a remote portion of the main memory, then an automatic request is sent out to fetch that line across the interconnection network, deliver it to the local bus, and then deliver it to the requesting cache on that bus. All of this activity is automatic and transparent to the processor and its cache.

In this configuration, cache coherence is a central concern. Although implementations differ as to details, in general terms we can say that each node must maintain some sort of directory that gives it an indication of the location of various portions of memory and also cache status information. To see how this scheme works, we give an example taken from [PFIS98]. Suppose that processor 3 on node 2 (P2-3) requests a memory location 798, which is in the memory of node 1. The following sequence occurs:

1. P2-3 issues a read request on the snoopy bus of node 2 for location 798.
2. The directory on node 2 sees the request and recognizes that the location is in node 1.
3. Node 2's directory sends a request to node 1, which is picked up by node 1's directory.
4. Node 1's directory, acting as a surrogate of P2-3, requests the contents of 798, as if it were a processor.
5. Node 1's main memory responds by putting the requested data on the bus.
6. Node 1's directory picks up the data from the bus.
7. The value is transferred back to node 2's directory.
8. Node 2's directory places the data back on node 2's bus, acting as a surrogate for the memory that originally held it.
9. The value is picked up and placed in P2-3's cache and delivered to P2-3.

The preceding sequence explains how data are read from a remote memory using hardware mechanisms that make the transaction transparent to the processor. On top of these mechanisms, some form of cache coherence protocol is needed. Various systems differ on exactly how this is done. We make only a few general remarks here. First, as part of the preceding sequence, node 1's directory keeps a record that some remote cache has a copy of the line containing location 798. Then, there needs to be a cooperative protocol to take care of modifications. For example, if a modification is done in a cache, this fact can be broadcast to other nodes. Each node's directory that receives such a broadcast can then determine if any local cache has that line and, if so, cause it to be purged. If the actual memory location is at the node receiving the broadcast notification, then that node's

directory needs to maintain an entry indicating that that line of memory is invalid and remains so until a write back occurs. If another processor (local or remote) requests the invalid line, then the local directory must force a write back to update memory before providing the data.

NUMA Pros and Cons

The main advantage of a CC-NUMA system is that it can deliver effective performance at higher levels of parallelism than SMP, without requiring major software changes. With multiple NUMA nodes, the bus traffic on any individual node is limited to a demand that the bus can handle. However, if many of the memory accesses are to remote nodes, performance begins to break down. There is reason to believe that this performance breakdown can be avoided. First, the use of L1 and L2 caches is designed to minimize all memory accesses, including remote ones. If much of the software has good temporal locality, then remote memory accesses should not be excessive. Second, if the software has good spatial locality, and if virtual memory is in use, then the data needed for an application will reside on a limited number of frequently used pages that can be initially loaded into the memory local to the running application. The Sequent designers report that such spatial locality does appear in representative applications [LOVE96]. Finally, the virtual memory scheme can be enhanced by including in the operating system a page migration mechanism that will move a virtual memory page to a node that is frequently using it; the Silicon Graphics designers report success with this approach [WHIT97].

Even if the performance breakdown due to remote access is addressed, there are two other disadvantages for the CC-NUMA approach [PFIS98]. First, a CC-NUMA does not transparently look like an SMP; software changes will be required to move an operating system and applications from an SMP to a CC-NUMA system. These include page allocation, already mentioned, process allocation, and load balancing by the operating system. A second concern is that of availability. This is a rather complex issue and depends on the exact implementation of the CC-NUMA system; the interested reader is referred to [PFIS98].

Vector Processor Simulator

17.7 CLOUD COMPUTING

Cloud computing was introduced in Chapter 1, where the three service models were discussed. Here we go into greater detail.

Cloud Computing Elements

NIST SP-800-145 (*The NIST Definition of Cloud Computing*) specifies that **cloud computing** is composed of five essential characteristics, three service models, and

four deployment models. Figure 17.12 illustrates the relationship among these concepts. The essential characteristics of cloud computing include the following:

- **Broad network access:** Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, laptops, and tablets) as well as other traditional or cloud-based software services.
- **Rapid elasticity:** Cloud computing gives you the ability to expand and reduce resources according to your specific service requirement. For example, you may need a large number of server resources for the duration of a specific task. You can then release these resources upon completion of the task.
- **Measured service:** Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.
- **On-demand self-service:** A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically

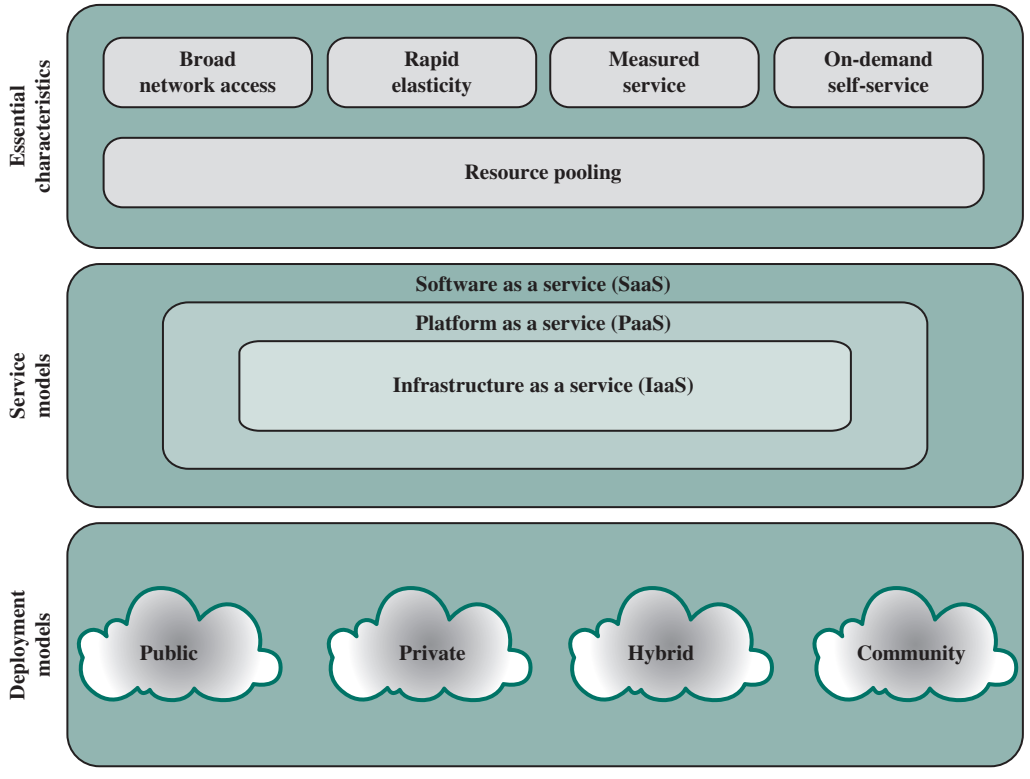


Figure 17.12 Cloud Computing Elements

without requiring human interaction with each service provider. Because the service is on demand, the resources are not permanent parts of your IT infrastructure.

- **Resource pooling:** The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a degree of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources, but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, network bandwidth, and virtual machines. Even private clouds tend to pool resources between different parts of the same organization.

NIST defines three **service models**, which can be viewed as nested service alternatives (Figure 17.13). These were defined in Chapter 1, and can be briefly summarized as follows:

- **Software as a service (SaaS):** Provides service to customers in the form of software, specifically application software, running on and accessible in the cloud.

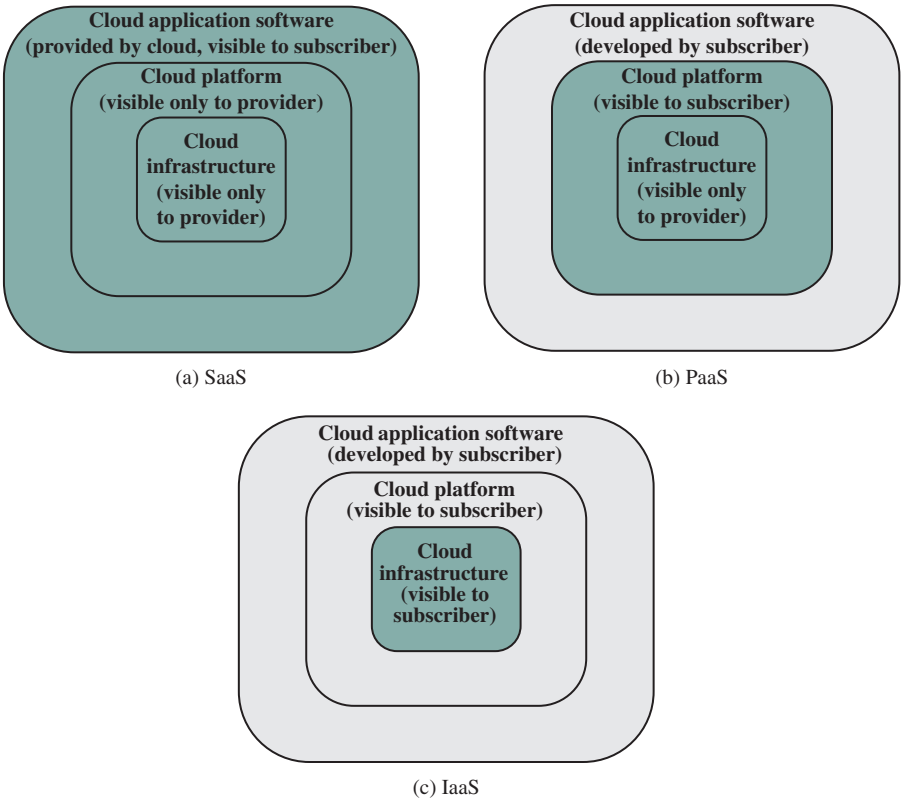


Figure 17.13 Cloud Service Models

- **Platform as a service (PaaS):** Provides service to customers in the form of a platform on which the customer's applications can run.
- **Infrastructure as a service (IaaS):** Provides the customer access to the underlying cloud infrastructure.

NIST defines four **deployment models**:

- **Public cloud:** The cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services. The cloud provider is responsible both for the cloud infrastructure and for the control of data and operations within the cloud. The major advantage of the public cloud is cost. A subscribing organization only pays for the services and resources it needs and can adjust these as needed. Further, the subscriber has greatly reduced management overhead. The principal concern is security. However, there are a number of public cloud providers that have demonstrated strong security controls and, in fact, such providers may have more resources and expertise to devote to security that would be available in a private cloud.
- **Private cloud:** A private cloud is a cloud infrastructure implemented within the internal IT environment of the organization. The organization may choose to manage the cloud in house or contract the management function to a third party. Additionally, the cloud servers and storage devices may exist on premise or off premise. A key motivation for opting for a private cloud is security. A private cloud infrastructure offers tighter controls over the geographic location of data storage and other aspects of security.
- **Community cloud:** A community cloud shares characteristics of private and public clouds. Like a private cloud, a community cloud is not open to any subscriber. Like a public cloud, the cloud resources are shared among a number of independent organizations. The organizations that share the community cloud have similar requirements and, typically, a need to exchange data with each other. One example of an industry that is employing the community cloud concept is the health care industry. A community cloud can be implemented to comply with government privacy and other regulations. The community participants can exchange data in a controlled fashion. The cloud infrastructure may be managed by the participating organizations or a third party and may exist on premise or off premise. In this deployment model, the costs are spread over fewer users than a public cloud (but more than a private cloud), so only some of the cost savings potential of cloud computing are realized.
- **Hybrid cloud:** The cloud infrastructure is a composition of two or more clouds (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds). With a hybrid cloud solution, sensitive information can be placed in a private area of the cloud, and less sensitive data can take advantage of the cost benefits of the public cloud.

Figure 17.14 illustrates the typical cloud service context. An enterprise maintains workstations within an enterprise LAN or set of LANs, which are connected by a router through a network or the Internet to the cloud service provider. The cloud service provider maintains a massive collection of servers, which it manages with a variety

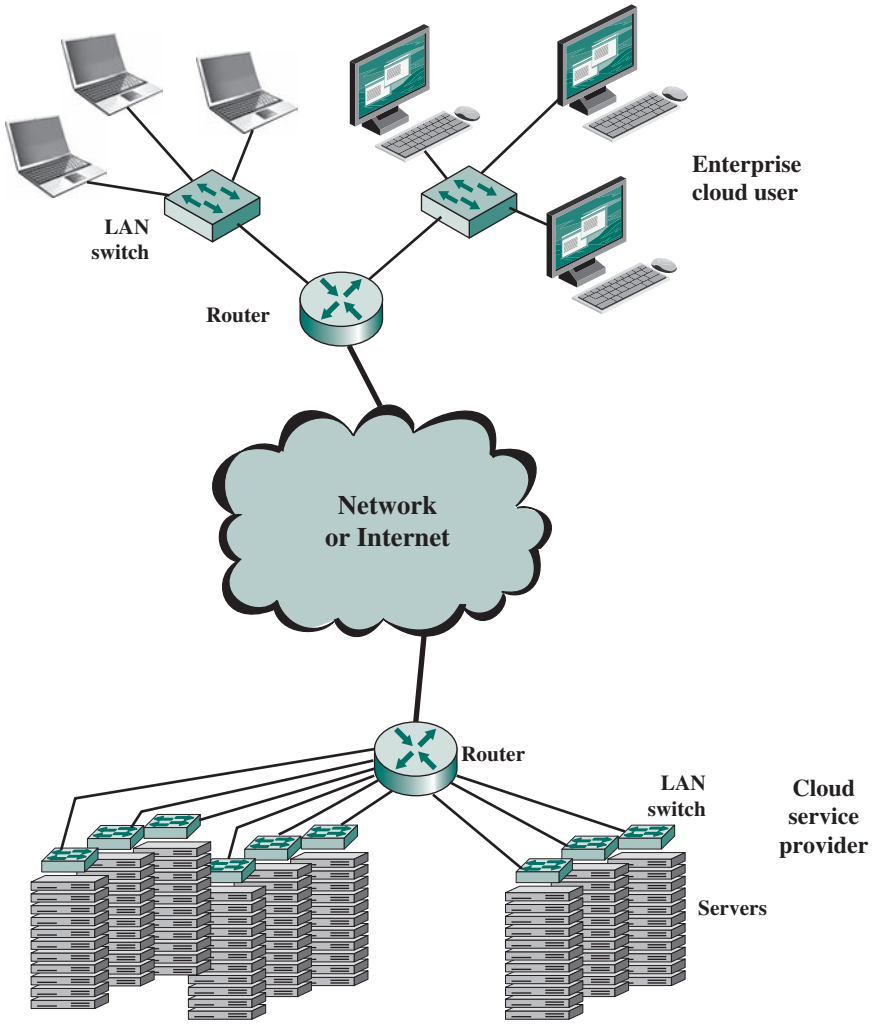


Figure 17.14 Cloud Computing Context

of network management, redundancy, and security tools. In the figure, the cloud infrastructure is shown as a collection of blade servers, which is a common architecture.

Cloud Computing Reference Architecture

NIST SP 500-292 (*NIST Cloud Computing Reference Architecture*) establishes a reference architecture, described as follows:

> The NIST cloud computing reference architecture focuses on the requirements of "what" cloud services provide, not a "how to" design solution and implementation. The reference architecture is intended to facilitate the understanding of the operational intricacies in cloud computing. It does not represent the system architecture of a specific cloud computing system; instead it is a tool for describing, discussing, and developing a system-specific architecture using a common framework of reference.

NIST developed the reference architecture with the following objectives in mind:

- To illustrate and understand the various cloud services in the context of an overall cloud computing conceptual model.
- To provide a technical reference for consumers to understand, discuss, categorize, and compare cloud services.
- To facilitate the analysis of candidate standards for security, interoperability, and portability and reference implementations.

The reference architecture, depicted in Figure 17.15, defines five major actors in terms of the roles and responsibilities:

- **Cloud consumer:** A person or organization that maintains a business relationship with, and uses service from, cloud providers.
- **Cloud provider (CP):** A person, organization, or entity responsible for making a service available to interested parties.
- **Cloud auditor:** A party that can conduct independent assessment of cloud services, information system operations, performance, and security of the cloud implementation.
- **Cloud broker:** An entity that manages the use, performance, and delivery of cloud services, and negotiates relationships between CPs and cloud consumers.
- **Cloud carrier:** An intermediary that provides connectivity and transport of cloud services from CPs to cloud consumers.

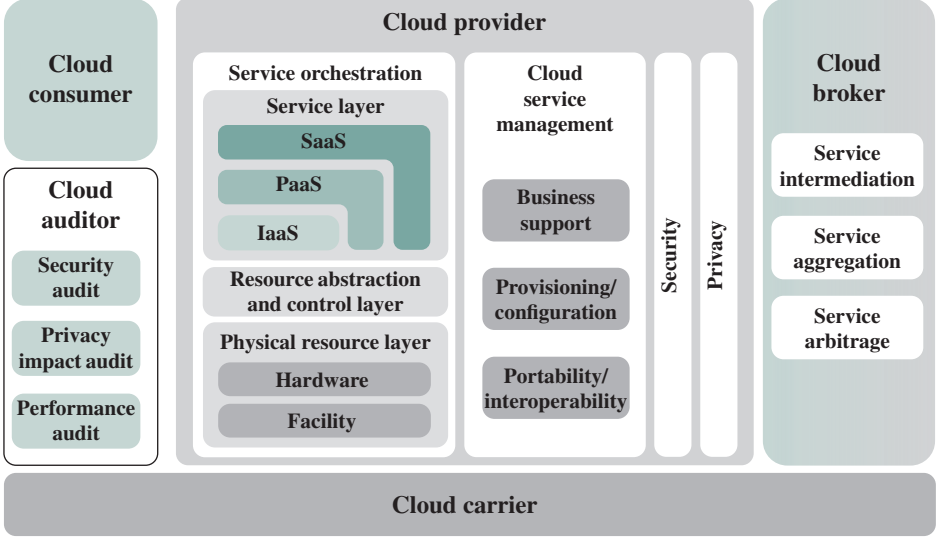


Figure 17.15 NIST Cloud Computing Reference Architecture

The roles of the cloud consumer and provider have already been discussed. To summarize, a **cloud provider** can provide one or more of the cloud services to meet IT and business requirements of **cloud consumers**. For each of the three service models (SaaS, PaaS, IaaS), the CP provides the storage and processing facilities needed to support that service model, together with a cloud interface for cloud service consumers. For SaaS, the CP deploys, configures, maintains, and updates the operation of the software applications on a cloud infrastructure so that the services are provisioned at the expected service levels to cloud consumers. The consumers of SaaS can be organizations that provide their members with access to software applications, end users who directly use software applications, or software application administrators who configure applications for end users.

For PaaS, the CP manages the computing infrastructure for the platform and runs the cloud software that provides the components of the platform, such as runtime software execution stack, databases, and other middleware components. Cloud consumers of PaaS can employ the tools and execution resources provided by CPs to develop, test, deploy, and manage the applications hosted in a cloud environment.

For IaaS, the CP acquires the physical computing resources underlying the service, including the servers, networks, storage, and hosting infrastructure. The IaaS cloud consumer in turn uses these computing resources, such as a virtual computer, for their fundamental computing needs.

The **cloud carrier** is a networking facility that provides connectivity and transport of cloud services between cloud consumers and CPs. Typically, a CP will set up service level agreements (SLAs) with a cloud carrier to provide services consistent with the level of SLAs offered to cloud consumers, and may require the cloud carrier to provide dedicated and secure connections between cloud consumers and CPs.

A **cloud broker** is useful when cloud services are too complex for a cloud consumer to easily manage. A cloud broker can offer three areas of support:

- **Service intermediation:** These are value-added services, such as identity management, performance reporting, and enhanced security.
- **Service aggregation:** The broker combines multiple cloud services to meet consumer needs not specifically addressed by a single CP, or to optimize performance or minimize cost.
- **Service arbitrage:** This is similar to service aggregation except that the services being aggregated are not fixed. Service arbitrage means a broker has the flexibility to choose services from multiple agencies. The cloud broker, for example, can use a credit-scoring service to measure and select an agency with the best score.

A **cloud auditor** can evaluate the services provided by a CP in terms of security controls, privacy impact, performance, and so on. The auditor is an independent entity that can assure that the CP conforms to a set of standards.

17.8 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

active standby	failback	private cloud
cache coherence	failover	public cloud
cluster	hybrid cloud	service aggregation
cloud auditor	infrastructure as a service (IaaS)	service arbitrage
cloud broker	MESI protocol	service intermediation
cloud carrier	multiprocessor	snoopy protocol
cloud computing	nonuniform memory access (NUMA)	software as a service (SaaS)
cloud consumer	passive standby	symmetric multiprocessor (SMP)
cloud provider	platform as a service (PaaS)	uniform memory access (UMA)
community cloud		uniprocessor
directory protocol		

Review Questions

17.1 List and briefly define three types of computer system organization.
17.2 What are the chief characteristics of an SMP?
17.3 What are some of the potential advantages of an SMP compared with a uniprocessor?
17.4 What are some of the key OS design issues for an SMP?
17.5 What is the difference between software and hardware cache coherent schemes?
17.6 What is the meaning of each of the four states in the MESI protocol?
17.7 What are some of the key benefits of clustering?
17.8 What is the difference between failover and failback?
17.9 What are the differences among UMA, NUMA, and CC-NUMA?
17.10 What is the cloud computing reference architecture?

Problems

17.1 Let α be the percentage of program code that can be executed simultaneously by n processors in a computer system. Assume that the remaining code must be executed sequentially by a single processor. Each processor has an execution rate of x MIPS.
- **a.** Derive an expression for the effective MIPS rate when using the system for exclusive execution of this program, in terms of n, α, and x.
- **b.** If $n = 16$ and $x = 4$ MIPS, determine the value of α that will yield a system performance of 40 MIPS.

17.2 A multiprocessor with eight processors has 20 attached tape drives. There are a large number of jobs submitted to the system that each require a maximum of four tape drives to complete execution. Assume that each job starts running with only three tape drives for a long period before requiring the fourth tape drive for a short period toward the end of its operation. Also assume an endless supply of such jobs.
- **a.** Assume the scheduler in the OS will not start a job unless there are four tape drives available. When a job is started, four drives are assigned immediately and are not released until the job finishes. What is the maximum number of jobs that can be in progress at once? What are the maximum and minimum number of tape drives that may be left idle as a result of this policy?

b. Suggest an alternative policy to improve tape drive utilization and at the same time avoid system deadlock. What is the maximum number of jobs that can be in progress at once? What are the bounds on the number of idling tape drives?

17.3 Can you foresee any problem with the write-once cache approach on bus-based multiprocessors? If so, suggest a solution.

17.4 Consider a situation in which two processors in an SMP configuration, over time, require access to the same line of data from main memory. Both processors have a cache and use the MESI protocol. Initially, both caches have an invalid copy of the line. Figure 17.16 depicts the consequence of a read of line x by Processor P1. If this is the start of a sequence of accesses, draw the subsequent figures for the following sequence:

1. P2 reads x.
2. P1 writes to x (for clarity, label the line in P1's cache x′).
3. P1 writes to x (label the line in P1's cache x″).
4. P2 reads x.

17.5 Figure 17.17 shows the state diagrams of two possible cache coherence protocols. Deduce and explain each protocol, and compare each to MESI.

17.6 Consider an SMP with both L1 and L2 caches using the MESI protocol. As explained in Section 17.3, one of four states is associated with each line in the L2 cache. Are all four states also needed for each line in the L1 cache? If so, why? If not, explain which state or states can be eliminated.

17.7 An earlier version of the IBM mainframe, the S/390 G4, used three levels of cache. As with the z990, only the first level was on the processor chip [called the processor unit (PU)]. The L2 cache was also similar to the z990. An L3 cache was on a separate chip that acted as a memory controller, and was interposed between the L2 caches and the memory cards. Table 17.3 shows the performance of a three-level cache arrangement for the IBM S/390. The purpose of this problem is to determine whether the inclusion of the third level of cache seems worthwhile. Determine the access penalty (average number of PU cycles) for a system with only an L1 cache, and normalize that value to 1.0. Then determine the normalized access penalty when both an L1 and L2 cache

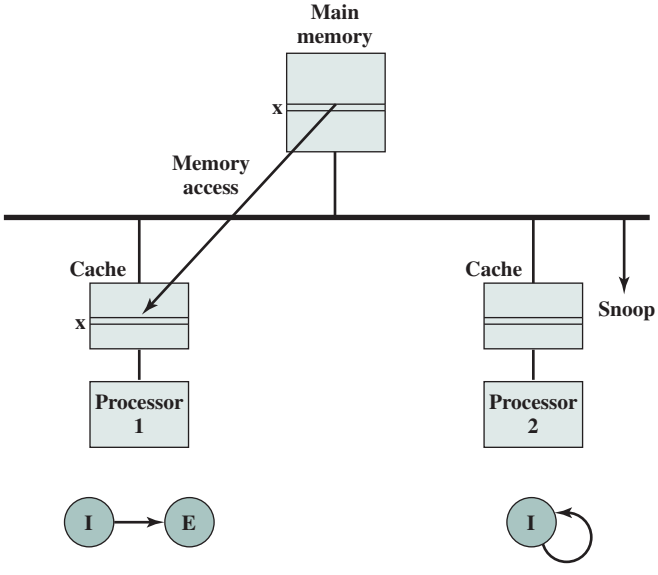


Figure 17.16 MESI Example: Processor 1 Reads Line x

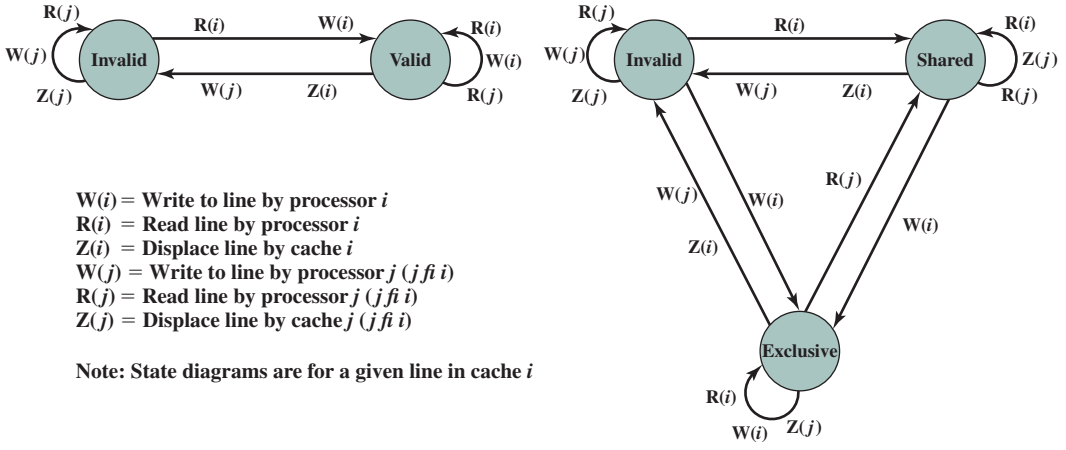


Figure 17.17 Two Cache Coherence Protocols

are used, and the access penalty when all three caches are used. Note the amount of improvement in each case and state your opinion on the value of the L3 cache.

17.8 **a.** Consider a uniprocessor with separate data and instruction caches, with hit ratios of H_d and H_i, respectively. Access time from processor to cache is c clock cycles, and transfer time for a block between memory and cache is b clock cycles. Let f_i be the fraction of memory accesses that are for instructions, and f_d is the fraction of dirty lines in the data cache among lines replaced. Assume a write-back policy and determine the effective memory access time in terms of the parameters just defined.

b. Now assume a bus-based SMP in which each processor has the characteristics of part (a). Every processor must handle cache invalidation in addition to memory reads and writes. This affects effective memory access time. Let f_{inv} be the fraction of data references that cause invalidation signals to be sent to other data caches. The processor sending the signal requires t clock cycles to complete the invalidation operation. Other processors are not involved in the invalidation operation. Determine the effective memory access time.

17.9 What organizational alternative is suggested by each of the illustrations in Figure 17.18?

17.10 In Figure 17.7, some of the diagrams show horizontal rows that are partially filled. In other cases, there are rows that are completely blank. These represent two different types of loss of efficiency. Explain.

17.11 Consider the pipeline depiction in Figure 14.13b, which is redrawn in Figure 17.19a, with the fetch and decode stages ignored, to represent the execution of thread A.

Table 17.3 Typical Cache Hit Rate on S/390 SMP Configuration [MAK97]

Memory Subsystem	Access Penalty (PU cycles)	Cache Size	Hit Rate (%)
L1 cache	1	32 KB	89
L2 cache	5	256 KB	5
L3 cache	14	2 MB	3
Memory	32	8 GB	3

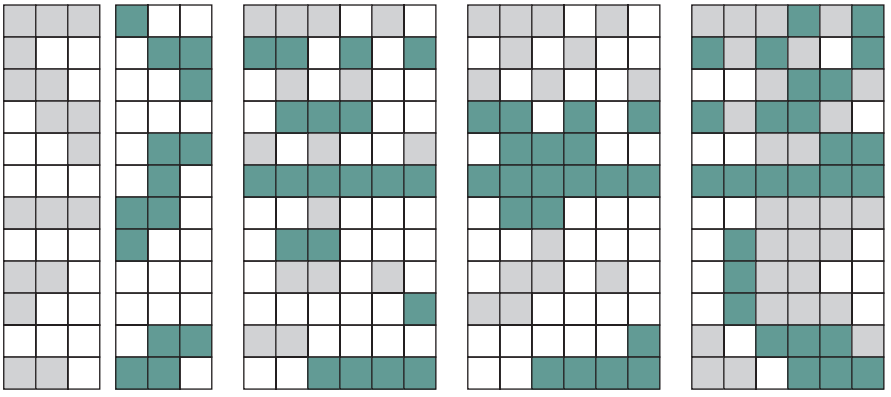


Figure 17.18 Diagram for Problem 17.9

Figure 17.19b illustrates the execution of a separate thread B. In both cases, a simple pipelined processor is used.

a. Show an instruction issue diagram, similar to Figure 17.7a, for each of the two threads.

b. Assume that the two threads are to be executed in parallel on a chip multiprocessor, with each of the two cores on the chip using a simple pipeline. Show an instruction issue diagram similar to Figure 17.7k. Also show a pipeline execution diagram in the style of Figure 17.19.

c. Assume a two-issue superscalar architecture. Repeat part (b) for an interleaved multithreading superscalar implementation, assuming no data dependencies.

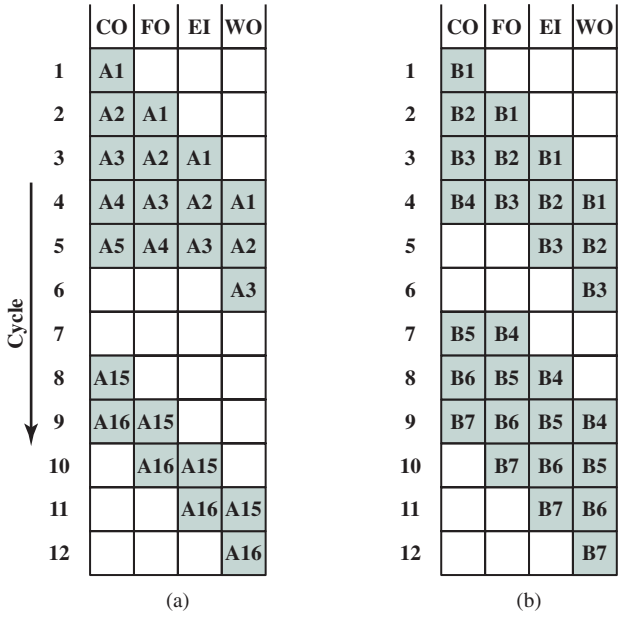


Figure 17.19 Two Threads of Execution

Note: There is no unique answer; you need to make assumptions about latency and priority.

d. Repeat part (c) for a blocked multithreading superscalar implementation.

e. Repeat for a four-issue SMT architecture.

17.12 An application program is executed on a nine-computer cluster. A benchmark program took time T on this cluster. Further, it was found that 25% of T was time in which the application was running simultaneously on all nine computers. The remaining time, the application had to run on a single computer.

a. Calculate the effective speedup under the aforementioned condition as compared to executing the program on a single computer. Also calculate α, the percentage of code that has been parallelized (programmed or compiled so as to use the cluster mode) in the preceding program.

b. Suppose that we are able to effectively use 17 computers rather than 9 computers on the parallelized portion of the code. Calculate the effective speedup that is achieved.

17.13 The following FORTRAN program is to be executed on a computer, and a parallel version is to be executed on a 32-computer cluster.

```
L1:      DO 10 I = 1, 1024
L2:            SUM(I) = 0
L3:            DO 20J = 1,I
L4:  20              SUM(I) = SUM(I) + I
L5:  10 CONTINUE
```

Suppose lines 2 and 4 each take two machine cycle times, including all processor and memory-access activities. Ignore the overhead caused by the software loop control statements (lines 1, 3, 5) and all other system overhead and resource conflicts.

a. What is the total execution time (in machine cycle times) of the program on a single computer?

b. Divide the I-loop iterations among the 32 computers as follows: Computer 1 executes the first 32 iterations (I = 1 to 32), processor 2 executes the next 32 iterations, and so on. What are the execution time and speedup factor compared with part (a)? (Note that the computational workload, dictated by the J-loop, is unbalanced among the computers.)

c. Explain how to modify the parallelizing to facilitate a balanced parallel execution of all the computational workload over 32 computers. By a balanced load is meant an equal number of additions assigned to each computer with respect to both loops.

d. What is the minimum execution time resulting from the parallel execution on 32 computers? What is the resulting speedup over a single computer?

17.14 Consider the following two versions of a program to add two vectors:

```
L1:      DO  10 I = 1, N
L2:       A(I) = B(I) + C(I)
L3: 10   CONTINUE
L4:       SUM = 0
L5:      DO 20J = 1, N
L6:       SUM = SUM + A(J)
L7: 20   CONTINUE
```

```
    DOALL  K = 1, M
     DO 10 I = L(K-1)+1, KL
      A(I) = B(I)+C(I)
10 CONTINUE
     SUM(K) = 0
    DO 20 J = 1, L
     SUM(K) = SUM(K)+A(L(K-1)+J)
20   CONTINUE
    ENDALL
```

a. The program on the left executes on a uniprocessor. Suppose each line of code L2, L4, and L6 takes one processor clock cycle to execute. For simplicity, ignore the time required for the other lines of code. Initially all arrays are already loaded in main memory and the short program fragment is in the instruction cache. How many clock cycles are required to execute this program?

b. The program on the right is written to execute on a multiprocessor with M processors. We partition the looping operations into M sections with $L = N/M$ elements per section. DOALL declares that all M sections are executed in parallel. The result of this program is to produce M partial sums. Assume that k clock cycles are needed for each interprocessor communication operation via the shared memory and that therefore the addition of each partial sum requires k cycles. An l-level binary adder tree can merge all the partial sums, where $l = \log_2 M$. How many cycles are needed to produce the final sum?

c. Suppose $N = 2^{20}$ elements in the array and $M = 256$. What is the speedup achieved by using the multiprocessor? Assume $k = 200$. What percentage is this of the theoretical speedup of a factor of 256?

CHAPTER 18

MULTICORE COMPUTERS

18.1 Hardware Performance Issues
- Increase in Parallelism and Complexity
- Power Consumption

18.2 Software Performance Issues
- Software on Multicore
- Application Example: Valve Game Software

18.3 Multicore Organization
- Levels of Cache
- Simultaneous Multithreading

18.4 Heterogeneous Multicore Organization
- Different Instruction Set Architectures
- Equivalent Instruction Set Architectures
- Cache Coherence and the MOESI Model

18.5 Intel Core i7-990X

18.6 ARM Cortex-A15 MPCore
- Interrupt Handling
- Cache Coherency
- L2 Cache Coherency

18.7 IBM zEnterprise EC12 Mainframe
- Organization
- Cache Structure

18.8 Key Terms, Review Questions, and Problems

Learning Objectives

After studying this chapter, you should be able to:

- Understand the **hardware performance issues** that have driven the move to multicore computers.
- Understand the **software performance issues** posed by the use of multithreaded multicore computers.
- Present an overview of the two principal approaches to **heterogeneous multicore organization**.
- Have an appreciation of the use of **multicore organization** on embedded systems, PCs and servers, and mainframes.

A **multicore processor**, also known as a **chip multiprocessor**, combines two or more processor units (called cores) on a single piece of silicon (called a die). Typically, each core consists of all of the components of an independent processor, such as registers, ALU, pipeline hardware, and control unit, plus L1 instruction and data caches. In addition to the multiple cores, contemporary multicore chips also include L2 cache and, increasingly, L3 cache. The most highly integrated multicore processors, known as systems on chip (SoCs), also include memory and peripheral controllers.

This chapter provides an overview of multicore systems. We begin with a look at the hardware performance factors that led to the development of multicore computers and the software challenges of exploiting the power of a multicore system. Next, we look at multicore organization. Finally, we examine three examples of multicore products, covering personal computer and workstation systems (Intel), embedded systems (ARM), and mainframes (IBM).

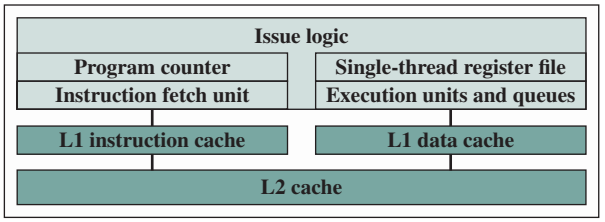
18.1 HARDWARE PERFORMANCE ISSUES

As we discuss in Chapter 2, microprocessor systems have experienced a steady increase in execution performance for decades. This increase is due to a number of factors, including increase in clock frequency, increase in transistor density, and refinements in the organization of the processor on the chip.

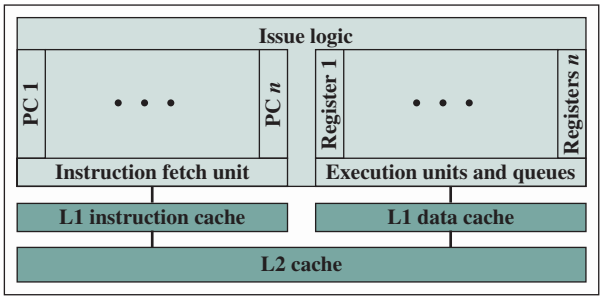
Increase in Parallelism and Complexity

The organizational changes in processor design have primarily been focused on exploiting ILP, so that more work is done in each clock cycle. These changes include, in chronological order (Figure 18.1):

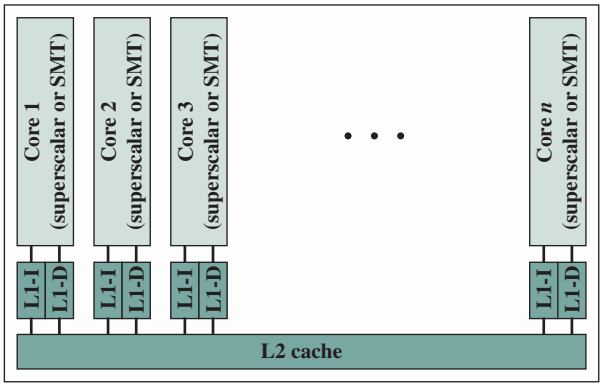
- **Pipelining:** Individual instructions are executed through a pipeline of stages so that while one instruction is executing in one stage of the pipeline, another instruction is executing in another stage of the pipeline.
- **Superscalar:** Multiple pipelines are constructed by replicating execution resources. This enables parallel execution of instructions in parallel pipelines, so long as hazards are avoided.



(a) Superscalar



(b) Simultaneous multithreading



(c) Multicore

Figure 18.1 Alternative Chip Organizations

- **Simultaneous multithreading (SMT):** Register banks are expanded so that multiple threads can share the use of pipeline resources.

With each of these innovations, designers have over the years attempted to increase the performance of the system by adding complexity. In the case of pipelining, simple three-stage pipelines were replaced by pipelines with five stages. Intel's Pentium 4 "Prescott" core had 31 stages for some instructions.

There is a practical limit to how far this trend can be taken, because with more stages, there is the need for more logic, more interconnections, and more control signals.

With superscalar organization, increased performance can be achieved by increasing the number of parallel pipelines. Again, there are diminishing returns

as the number of pipelines increases. More logic is required to manage hazards and to stage instruction resources. Eventually, a single thread of execution reaches the point where hazards and resource dependencies prevent the full use of the multiple pipelines available. Also, compiled binary code rarely exposes enough ILP to take advantage of more than about six parallel pipelines.

This same point of diminishing returns is reached with SMT, as the complexity of managing multiple threads over a set of pipelines limits the number of threads and number of pipelines that can be effectively utilized. SMT's advantage lies in the fact that two (or more) program streams can be searched for available ILP.

There is a related set of problems dealing with the design and fabrication of the computer chip. The increase in complexity to deal with all of the logical issues related to very long pipelines, multiple superscalar pipelines, and multiple SMT register banks means that increasing amounts of the chip area are occupied with coordinating and signal transfer logic. This increases the difficulty of designing, fabricating, and debugging the chips. The increasingly difficult engineering challenge related to processor logic is one of the reasons that an increasing fraction of the processor chip is devoted to the simpler memory logic. Power issues, discussed next, provide another reason.

Power Consumption

To maintain the trend of higher performance as the number of transistors per chip rises, designers have resorted to more elaborate processor designs (pipelining, superscalar, SMT) and to high clock frequencies. Unfortunately, power requirements have grown exponentially as chip density and clock frequency have risen. This was shown in Figure 2.2.

One way to control power density is to use more of the chip area for cache memory. Memory transistors are smaller and have a power density an order of magnitude lower than that of logic (see Figure 18.2). As chip transistor density has increased, the percentage of chip area devoted to memory has grown, and is now often half the chip area. Even so, there is still a considerable amount of chip area devoted to processing logic.

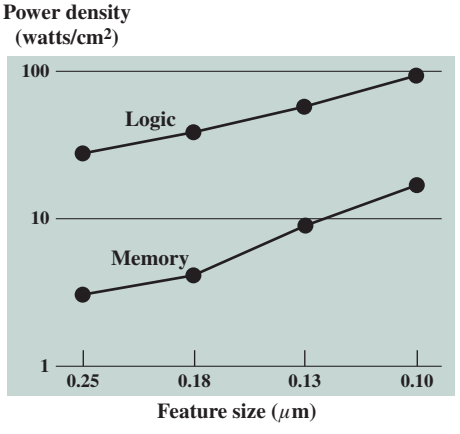


Figure 18.2 Power and Memory Considerations

How to use all those logic transistors is a key design issue. As discussed earlier in this section, there are limits to the effective use of such techniques as superscalar and SMT. In general terms, the experience of recent decades has been encapsulated in a rule of thumb known as **Pollack's rule** [POLL99], which states that performance increase is roughly proportional to square root of increase in complexity. In other words, if you double the logic in a processor core, then it delivers only 40% more performance. In principle, the use of multiple cores has the potential to provide near-linear performance improvement with the increase in the number of cores—but only for software that can take advantage.

Power considerations provide another motive for moving toward a multicore organization. Because the chip has such a huge amount of cache memory, it becomes unlikely that any one thread of execution can effectively use all that memory. Even with SMT, multithreading is done in a relatively limited fashion and cannot therefore fully exploit a gigantic cache, whereas a number of relatively independent threads or processes has a greater opportunity to take full advantage of the cache memory.

18.2 SOFTWARE PERFORMANCE ISSUES

A detailed examination of the software performance issues related to multicore organization is beyond our scope. In this section, we first provide an overview of these issues, and then look at an example of an application designed to exploit multicore capabilities.

Software on Multicore

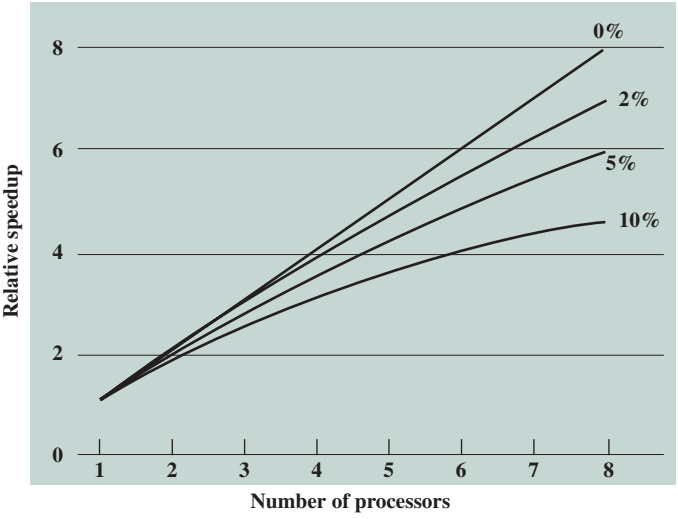
The potential performance benefits of a multicore organization depend on the ability to effectively exploit the parallel resources available to the application. Let us focus first on a single application running on a multicore system. Recall from Chapter 2 that Amdahl's law states that:

$$\text{Speed up} = \frac{\text{time to execute program on a single processor}}{\text{time to execute program on } N \text{ parallel processors}}$$

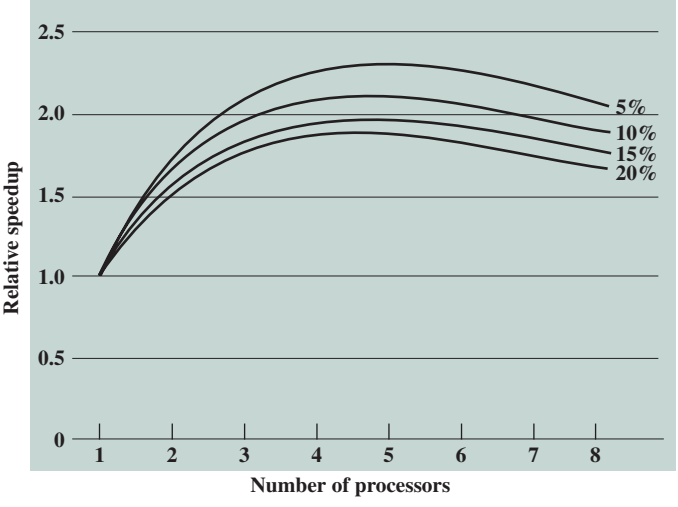
$$= \frac{1}{(1 - -f) + \frac{f}{N}} \tag{18.1}$$

The law assumes a program in which a fraction $(1 - -f)$ of the execution time involves code that is inherently sequential and a fraction f that involves code that is infinitely parallelizable with no scheduling overhead.

This law appears to make the prospect of a multicore organization attractive. But as Figure 18.3a shows, even a small amount of serial code has a noticeable impact. If only 10% of the code is inherently serial ($f = 0.9$), running the program on a multicore system with eight processors yields a performance gain of only a factor of 4.7. In addition, software typically incurs overhead as a result of communication and distribution of work among multiple processors and as a result of cache



(a) Speedup with 0%, 2%, 5%, and 10% sequential portions



(b) Speedup with overheads

Figure 18.3 Performance Effect of Multiple Cores

coherence overhead. This overhead results in a curve where performance peaks and then begins to degrade because of the increased burden of the overhead of using multiple processors (e.g., coordination and OS management). Figure 18.3b, from [MCDO05], is a representative example.

However, software engineers have been addressing this problem and there are numerous applications in which it is possible to effectively exploit a multicore system. [MCDO05] analyzes the effectiveness of multicore systems on a set of database applications, in which great attention was paid to reducing the serial fraction within hardware architectures, operating systems, middleware, and the database

application software. Figure 18.4 shows the result. As this example shows, database management systems and database applications are one area in which multicore systems can be used effectively. Many kinds of servers can also effectively use the parallel multicore organization, because servers typically handle numerous relatively independent transactions in parallel.

In addition to general-purpose server software, a number of classes of applications benefit directly from the ability to scale throughput with the number of cores. [MCDO06] lists the following examples:

- **Multithreaded native applications (thread-level parallelism):** Multithreaded applications are characterized by having a small number of highly threaded processes.
- **Multiprocess applications (process-level parallelism):** Multiprocess applications are characterized by the presence of many single-threaded processes.
- **Java applications:** Java applications embrace threading in a fundamental way. Not only does the Java language greatly facilitate multithreaded applications, but the Java Virtual Machine is a multithreaded process that provides scheduling and memory management for Java applications.
- **Multi-instance applications (application-level parallelism):** Even if an individual application does not scale to take advantage of a large number of threads, it is still possible to gain from multicore architecture by running multiple instances of the application in parallel. If multiple application instances require some degree of isolation, virtualization technology (for the hardware of the operating system) can be used to provide each of them with its own separate and secure domain.

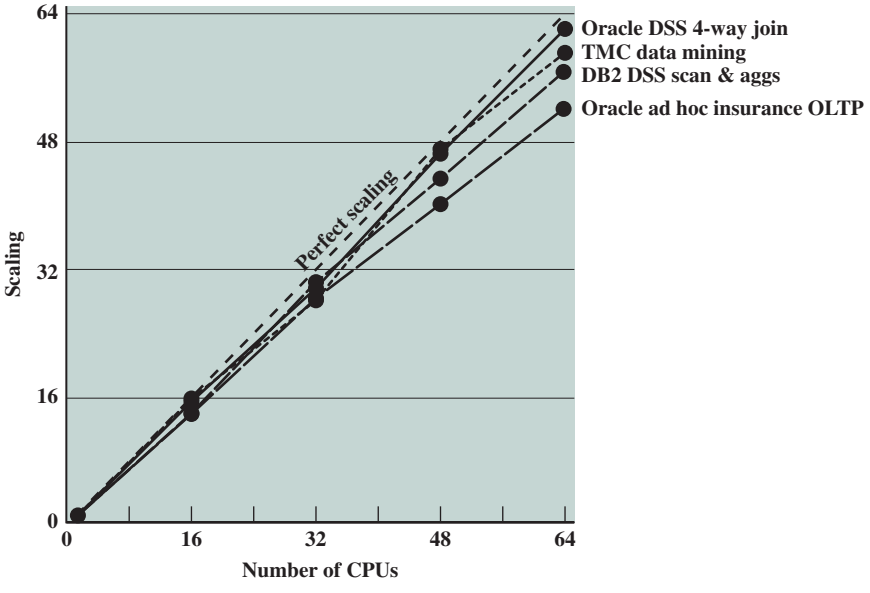


Figure 18.4 Scaling of Database Workloads on Multiple-Processor Hardware

Before turning to an example, we elaborate on the topic of thread-level parallelism by introducing the concept of **threading granularity**, which can be defined as the minimal unit of work that can be beneficially parallelized. In general, the finer the granularity the system enables, the less constrained is the programmer in parallelizing a program. Consequently, finer grain threading systems allow parallelization in more situations than coarse-grained ones. The choice of the target granularity of an architecture involves an inherent tradeoff. On the one hand, the finer grain systems are preferable because of the flexibility they afford to the programmer. On the other hand, the finer the threading granularity, the more significant part of the execution is taken by the threading system overhead.

Application Example: Valve Game Software

Valve is an entertainment and technology company that has developed a number of popular games as well as the Source engine, one of the most widely played game engines available. Source is an animation engine used by Valve for its games and licensed to other game developers.

Valve has reprogrammed the Source engine software to use multithreading to exploit the scalability of multicore processor chips from Intel and AMD [REIM06]. The revised Source engine code provides more powerful support for Valve games such as Half Life 2.

From Valve's perspective, threading granularity options are defined as follows [HARR06]:

- **Coarse-grained threading:** Individual modules, called systems, are assigned to individual processors. In the Source engine case, this means putting rendering on one processor, AI (artificial intelligence) on another, physics on another, and so on. This is straightforward. In essence, each major module is single threaded and the principal coordination involves synchronizing all the threads with a timeline thread.
- **Fine-grained threading:** Many similar or identical tasks are spread across multiple processors. For example, a loop that iterates over an array of data can be split up into a number of smaller parallel loops in individual threads that can be scheduled in parallel.
- **Hybrid threading:** This involves the selective use of fine-grain threading for some systems and single threading for other systems.

Valve found that through coarse threading, it could achieve up to twice the performance across two processors compared to executing on a single processor. But this performance gain could only be achieved with contrived cases. For real-world gameplay, the improvement was on the order of a factor of 1.2. Valve also found that effective use of fine-grain threading was difficult. The time per work unit can be variable, and managing the timeline of outcomes and consequences involved complex programming.

Valve found that a hybrid threading approach was the most promising and would scale the best as multicore systems with eight or sixteen processors became available. Valve identified systems that operate very effectively when assigned to a single processor permanently. An example is sound mixing, which has little user interaction, is not constrained by the frame configuration of windows, and works on

its own set of data. Other modules, such as scene rendering, can be organized into a number of threads so that the module can execute on a single processor but achieve greater performance as it is spread out over more and more processors.

Figure 18.5 illustrates the thread structure for the rendering module. In this hierarchical structure, higher-level threads spawn lower-level threads as needed. The rendering module relies on a critical part of the Source engine, the world list, which is a database representation of the visual elements in the game's world. The first task is to determine what are the areas of the world that need to be rendered. The next task is to determine what objects are in the scene as viewed from multiple angles. Then comes the processor-intensive work. The rendering module has to work out the rendering of each object from multiple points of view, such as the player's view, the view of TV monitors, and the point of view of reflections in water.

Some of the key elements of the threading strategy for the rendering module are listed in [LEON07] and include the following:

- Construct scene-rendering lists for multiple scenes in parallel (e.g., the world and its reflection in water).
- Overlap graphics simulation.
- Compute character bone transformations for all characters in all scenes in parallel.
- Allow multiple threads to draw in parallel.

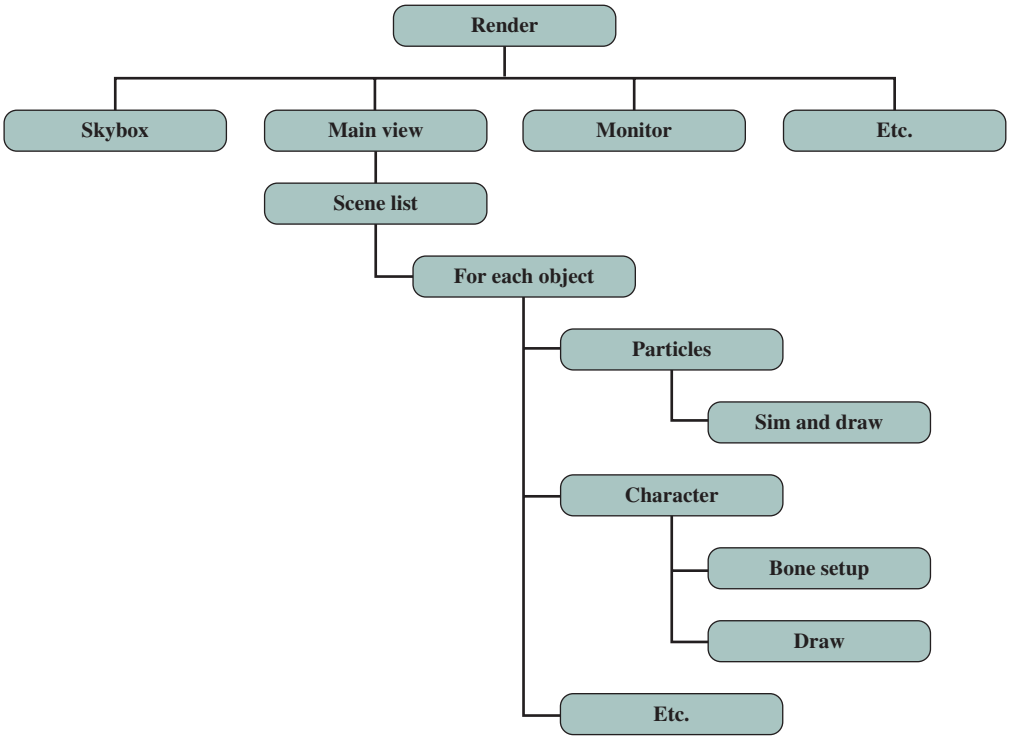


Figure 18.5 Hybrid Threading for Rendering Module

The designers found that simply locking key databases, such as the world list, for a thread was too inefficient. Over 95% of the time, a thread is trying to read from a data set, and only 5% of the time at most is spent in writing to a data set. Thus, a concurrency mechanism known as the single-writer-multiple-readers model works effectively.

18.3 MULTICORE ORGANIZATION

At a top level of description, the main variables in a multicore organization are as follows:

- The number of core processors on the chip
- The number of levels of cache memory
- How cache memory is shared among cores
- Whether simultaneous multithreading (SMT) is employed
- The types of cores

We explore all but the last of these considerations in this section, deferring a discussion of types of cores to the next section.

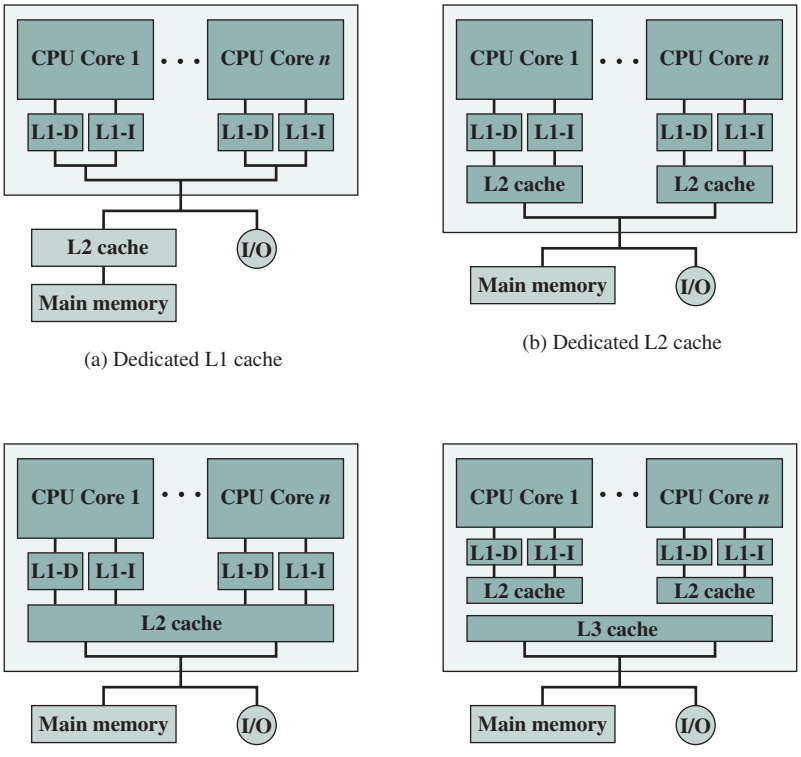
Levels of Cache

Figure 18.6 shows four general organizations for multicore systems. Figure 18.6a is an organization found in some of the earlier multicore computer chips and is still seen in some embedded chips. In this organization, the only on-chip cache is L1 cache, with each core having its own dedicated L1 cache. Almost invariably, the L1 cache is divided into instruction and data caches for performance reasons, while L2 and higher-level caches are unified. An example of this organization is the ARM11 MPCore.

The organization of Figure 18.6b is also one in which there is no on-chip cache sharing. In this, there is enough area available on the chip to allow for L2 cache. An example of this organization is the AMD Opteron. Figure 18.6c shows a similar allocation of chip space to memory, but with the use of a shared L2 cache. The Intel Core Duo has this organization. Finally, as the amount of cache memory available on the chip continues to grow, performance considerations dictate splitting off a separate, shared L3 cache (Figure 18.6d), with dedicated L1 and L2 caches for each core processor. The Intel Core i7 is an example of this organization.

The use of a shared higher-level cache on the chip has several advantages over exclusive reliance on dedicated caches:

1. Constructive interference can reduce overall miss rates. That is, if a thread on one core accesses a main memory location, this brings the line containing the referenced location into the shared cache. If a thread on another core soon thereafter accesses the same memory block, the memory locations will already be available in the shared on-chip cache.
2. A related advantage is that data shared by multiple cores is not replicated at the shared cache level.



(a) Dedicated L1 cache

(b) Dedicated L2 cache

(c) Shared L2 cache

(d) Shared L3 cache

Figure 18.6 Multicore Organization Alternatives

3. With proper line replacement algorithms, the amount of shared cache allocated to each core is dynamic, so that threads that have less locality (larger working sets) can employ more cache.
4. Inter-core communication is easy to implement, via shared memory locations.
5. The use of a shared higher-level cache confines the cache coherency problem to the lower cache levels, which may provide some additional performance advantage.

A potential advantage to having only dedicated L2 caches on the chip is that each core enjoys more rapid access to its private L2 cache. This is advantageous for threads that exhibit strong locality.

As both the amount of memory available and the number of cores grow, the use of a shared L3 cache combined with dedicated percore L2 caches seems likely to provide better performance than simply a massive shared L2 cache or very large dedicated L2 caches with no on-chip L3. An example of this latter arrangement is the Xeon E5-2600/4600 chip processor (Figure 7.1)

Not shown is the arrangement where L1s are local to each core, L2s are shared among 2 to 4 cores, and L3 is global across all cores. This arrangement is likely to become more common over time.

Simultaneous Multithreading

Another organizational design decision in a multicore system is whether the individual cores will implement **simultaneous multithreading (SMT)**. For example, the Intel Core Duo uses pure superscalar cores, whereas the Intel Core i7 uses SMT cores. SMT has the effect of scaling up the number of hardware-level threads that the multicore system supports. Thus, a multicore system with four cores and SMT that supports four simultaneous threads in each core appears the same to the application level as a multicore system with 16 cores. As software is developed to more fully exploit parallel resources, an SMT approach appears to be more attractive than a purely superscalar approach.

18.4 HETEROGENEOUS MULTICORE ORGANIZATION

The quest to make optimal use of the silicon real estate on a processor chip is never ending. As clock speeds and logic densities increase, designers must balance many design elements in attempts to maximize performance and minimize power consumption. We have so far examined a number of such approaches, including the following:

1. Increase the percentage of the chip devoted to cache memory.
2. Increase the number of levels of cache memory.
3. Change the length (increase or decrease) and functional components of the instruction pipeline.
4. Employ simultaneous multithreading.
5. Use multiple cores.

A typical case for the use of multiple cores is a chip with multiple identical cores, known as **homogenous multicore organization**. To achieve better results, in terms of performance and/or power consumption, an increasingly popular design choice is **heterogeneous multicore organization**, which refers to a processor chip that includes more than one kind of core. In this section, we look at two approaches to heterogeneous multicore organization.

Different Instruction Set Architectures

The approach that has received the most industry attention is the use of cores that have distinct ISAs. Typically, this involves mixing conventional cores, referred to in this context as CPUs, with specialized cores optimized for certain types of data or applications. Most often, the additional cores are optimized to deal with vector and matrix data processing.

CPU/GPU MULTICORE The most prominent trend in terms of heterogeneous multicore design is the use of both CPUs and graphics processing units (GPUs) on the same chip. GPUs are discussed in detail in the following chapter. Briefly, GPUs are characterized by the ability to support thousands of parallel execution threads. Thus, GPUs are well matched to applications that process large amounts

of vector and matrix data. Initially aimed at improving the performance of graphics applications, thanks to easy-to-adopt programming models such as CUDA (Compute Unified Device Architecture), these new processors are increasingly being applied to improve the performance of general-purpose and scientific applications that involve large numbers of repetitive operations on structured data.

To deal with the diversity of target applications in today's computing environment, multicore containing both GPUs and CPUs has the potential to enhance performance. This heterogeneous mix, however, presents issues of coordination and correctness.

Figure 18.7 is a typical multicore processor organization. Multiple CPUs and GPUs share on-chip resources, such as the last-level cache (LLC), interconnection network, and memory controllers. Most critical is the way in which cache management policies provide effective sharing of the LLC. The differences in cache sensitivity and memory access rate between CPUs and GPUs create significant challenges to the efficient sharing of the LLC.

Table 18.1 illustrates the potential performance benefit of combining CPUs and GPUs for scientific applications. This table shows the basic operating parameters of an AMD chip, the A10 5800K [ALTS12]. For floating-point calculations, the CPU's performance at 121.6 GFLOPS is dwarfed by the GPU, which offers 614 GFLOPS to applications that can utilize the resource effectively.

Whether it is scientific applications or traditional graphics processing, the key to leveraging the added GPU processors is to consider the time needed to transfer a block of data to the GPU, process it, then return the results to the main application thread. In earlier implementations of chips that incorporated GPUs, physical memory is partitioned between CPU and GPU. If an application thread is running on a CPU that demands GPU processing, the CPU explicitly copies the data to the GPU memory. The GPU completes the computation and then copies the result back to CPU memory. Issues of cache coherence across CPU and GPU memory caches do not arise because the memory is partitioned. On the other hand, the physical handing of data back and forth results in a performance penalty.

A number of research and development efforts are underway to improve performance over that described in the preceding paragraph, of which the most notable

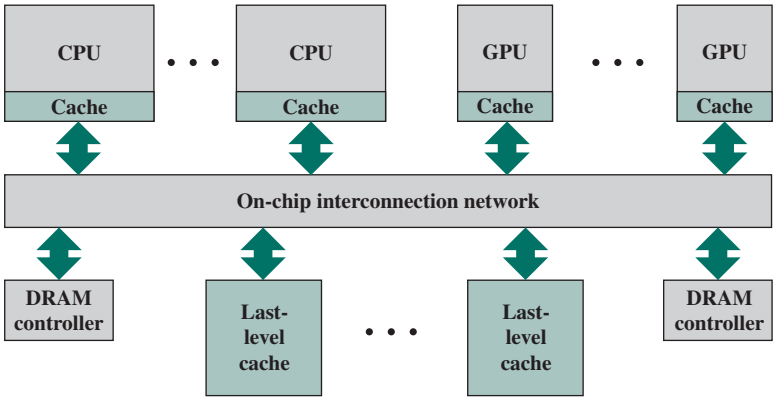


Figure 18.7 Heterogenous Multicore Chip Elements

Table 18.1 Operating Parameters of AMD 5100K Heterogeneous Multicore Processor

	CPU	GPU
Clock frequency (GHz)	3.8	0.8
Cores	4	384
FLOPS/core	8	2
GFLOPS	121.6	614.4

FLOPS = floating-point operations per second.
FLOPS/core = number of parallel floating-point operations that can be performed.

is the initiative by the Heterogeneous System Architecture (HSA) Foundation. Key features of the HSA approach include the following:

1. The entire virtual memory space is visible to both CPU and GPU. Both CPU and GPU can access and allocate any location in the system's virtual memory space.
2. The virtual memory system brings in pages to physical main memory as needed.
3. A coherent memory policy ensures that CPU and GPU caches both see an up-to-date view of data.
4. A unified programming interface that enables users to exploit the parallel capabilities of the GPUs within programs that rely on CPU execution as well.

The overall objective is to allow programmers to write applications that exploit the serial power of CPUs and the parallel-processing power of GPUs seamlessly with efficient coordination at the OS and hardware level. As mentioned, this is an ongoing area of research and development.

CPU/DSP MULTICORE Another common example of a heterogeneous multicore chip is a mixture of CPUs and digital signal processors (DSPs). A DSP provides ultra-fast instruction sequences (shift and add; multiply and add), which are commonly used in math-intensive digital signal processing applications. DSPs are used to process analog data from sources such as sound, weather satellites, and earthquake monitors. Signals are converted into digital data and analyzed using various algorithms such as Fast Fourier Transform. DSP cores are widely used in myriad devices, including cellphones, sound cards, fax machines, modems, hard disks, and digital TVs.

As a good representative example, Figure 18.8 shows a recent version of Texas Instruments (TI) K2H SoC platform [TI12]. This heterogeneous multicore processor delivers power-efficient processing solutions for high-end imaging applications. TI lists the performance as delivering up to 352 GMACS, 198 GFLOPS, and 19,600 MIPS. GMACS stands for giga (billions of) multiply-accumulate operations per second, a common measure of DSP performance. Target applications for these systems include industrial automation, video surveillance, high-end inspection systems, industrial printers/scanners, and currency/counterfeit detection.

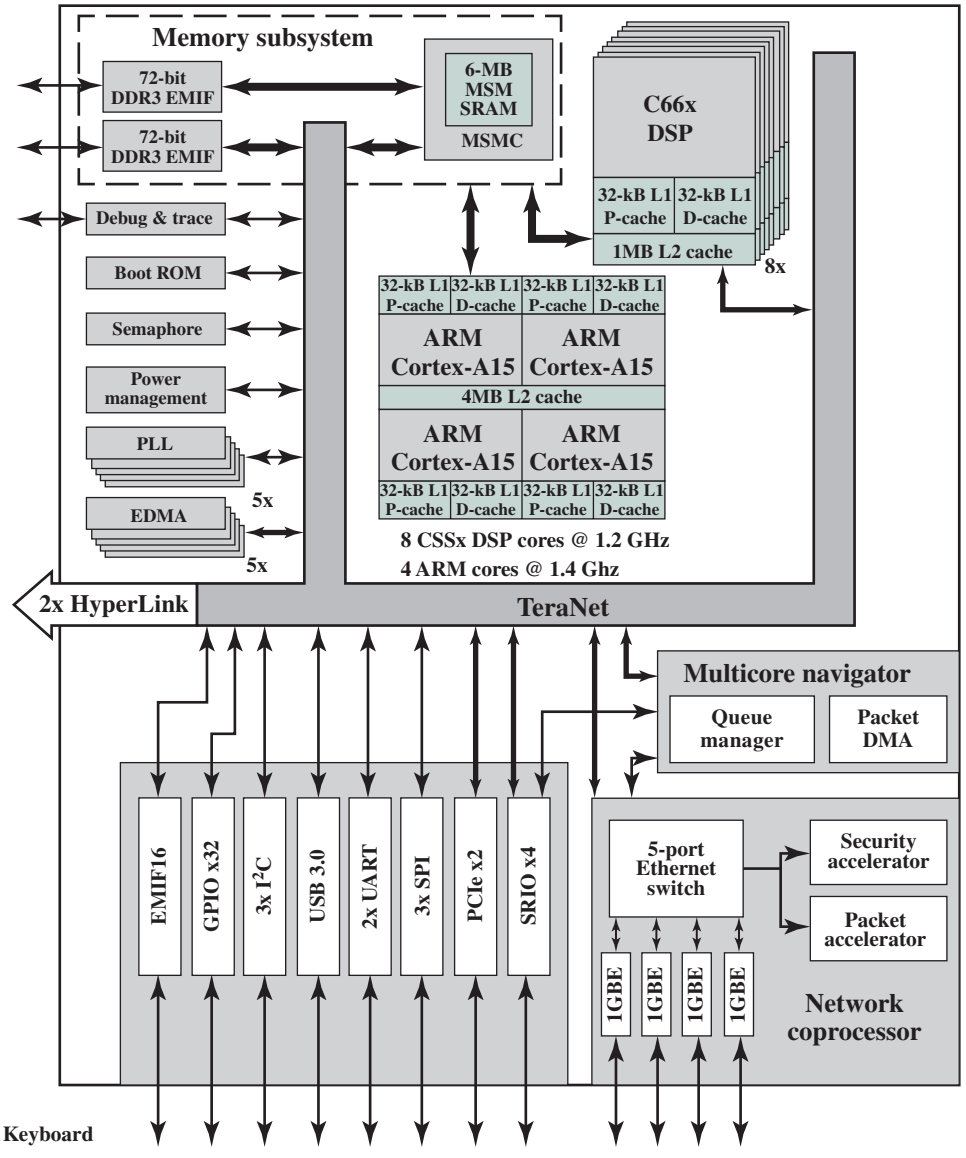


Figure 18.8 Texas Instruments 66AK2H12 Heterogenous Multicore Chip

The TI chip includes four ARM Cortex-A15 cores and eight TI C66x DSP cores.

Each DSP core contains 32 kB of L1 data cache and 32 kB of L1 program (instruction) cache. In addition, each DSP has 1 MB of dedicated SRAM memory that can be configured as all L2 cache, all main memory, or a mix of the two. The portion configured as main memory functions as a "local" main memory, referred to simply as *SRAM*. This local main memory can be used for temporary data, avoiding the need for traffic between cache and off-chip memory. The L2 cache of each of

the eight DSP cores is dedicated rather than shared with the other DSP cores. This is typical for a multicore DSP organization: Each DSP works on a separate block of data in parallel, so there is little need for data sharing.

Each ARM Cortex-A15 CPU core has 32-kB L1 data and program caches, and the four cores share a 4-MB L2 cache.

The 6-MB multicore shared memory (MSM) is always configured as all SRAM. That is, it behaves like main memory rather than cache. It can be configured to feed directly the L1 DSP and CPU caches, or to feed the L2 DSP and CPU caches. This configuration decision depends on the expected application profile. The multicore shared memory controller (MSMC) manages traffic among ARM cores, DSP, DMA, other mastering peripherals, and the external memory interface (EMIF). MSMC controls access to the MSM, which is accessible by all the cores and the mastering peripherals on the device.

Equivalent Instruction Set Architectures

Another recent approach to heterogeneous multicore organization is the use of multiple cores that have equivalent ISAs but vary in performance or power efficiency. The leading example of this is ARM's big.Little architecture, which we examine in this section.

Figure 18.9 illustrates this architecture. The figure shows a multicore processor chip containing two high-performance Cortex-A15 cores and two lower-performance, lower-power-consuming Cortex-A7 cores. The A7 cores handle less computation-intense tasks, such as background processing, playing music, sending texts, and making phone calls. The A15 cores are invoked for high intensity tasks, such as for video, gaming, and navigation.

The big.Little architecture is aimed at the smartphone and tablet market. These are devices whose performance demands from users are increasing at a much faster rate than the capacity of batteries or the power savings from semiconductor process advances. The usage pattern for smartphones and tablets is quite dynamic. Periods of processing-intense tasks, such as gaming and web browsing, alternate

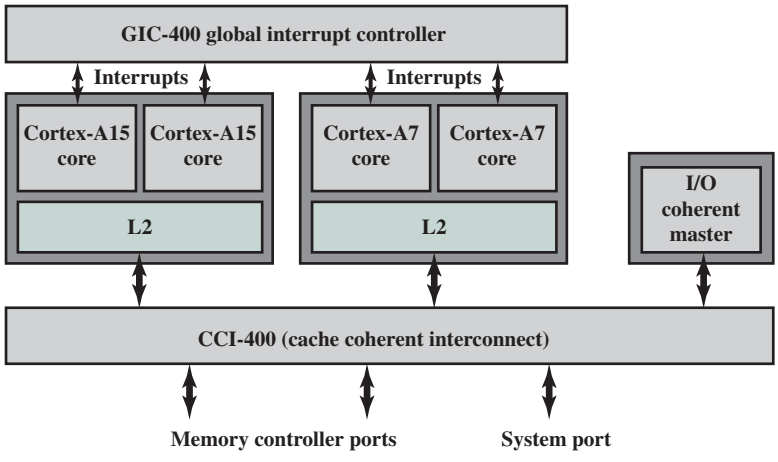
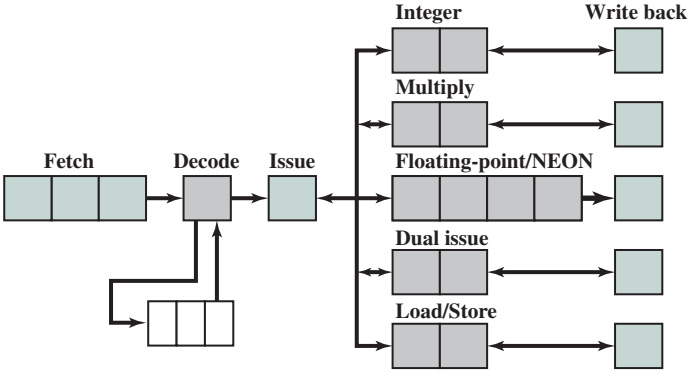


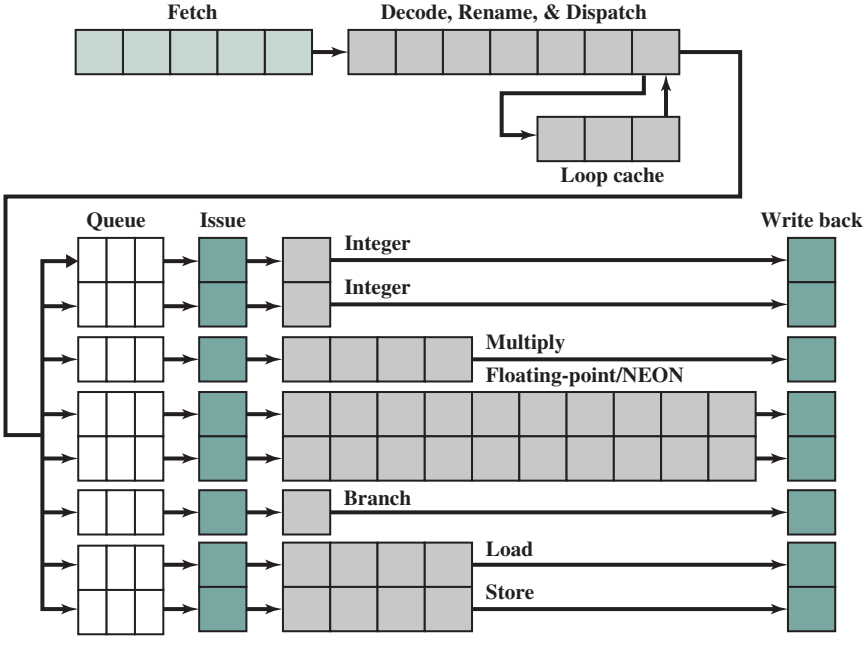
Figure 18.9 big.Little Chip Components

with typically longer periods of low processing-intensity tasks, such as texting, e-mail, and audio. The big.Little architecture takes advantage of this variation in required performance. The A15 is designed for maximum performance within the mobile power budget. The A7 processor is designed for maximum efficiency and high enough performance to address all but the most intense periods of work.

A7 AND A15 CHARACTERISTICS The A7 is far simpler and less powerful than the A15. But its simplicity requires far fewer transistors than does the A15's complexity—and fewer transistors require less energy to operate. The differences between the A7 and A15 cores are seen most clearly by examining their instruction pipelines, as shown in Figure 18.10.



(a) Cortex A-7 Pipeline



(b) Cortex A-15 Pipeline

Figure 18.10 Cortex A-7 and A-15 Pipelines

The A7 is an in-order CPU with a pipeline length of 8 to 10 stages. It has a single queue for all of its execution units, and two instructions can be sent to its five execution units per clock cycle. The A15, on the other hand, is an out-of-order processor with a pipeline length of 15 to 24 stages. Each of its eight execution units has its own multistage queue, and three instructions can be processed per clock cycle.

The energy consumed by the execution of an instruction is partially related to the number of pipeline stages it must traverse. Therefore, a significant difference in energy consumption between Cortex-A15 and Cortex-A7 comes from the different pipeline complexity. Across a range of benchmarks, the Cortex-A15 delivers roughly twice the performance of the Cortex-A7 per unit MHz, and the Cortex-A7 is roughly three times as energy efficient as the Cortex-A15 in completing the same workloads [JEFF12]. The performance tradeoff is illustrated in Figure 18.11 [STEV13].

SOFTWARE PROCESSING MODELS The big.Little architecture can be configured to use one of two software processing models: migration and multiprocessing (MP). The software models differ mainly in the way they allocate work to big or Little cores during runtime execution of a workload.

In the migration model, big and Little cores are paired. To the OS kernel scheduler, each big/Little pair is visible as a single core. Power management software is responsible for migrating software contexts between the two cores. This model is a natural extension to the dynamic voltage and frequency scaling (DVFS) operating points provided by current mobile platforms to allow the OS to match the performance of the platform to the performance required by the application. In today's smartphone SoCs, DVFS drivers like cpu_freq sample the OS performance at regular and frequent intervals, and the DVFS governor decides whether to shift to a higher or lower operating point or remain at the current operating point. As shown in Figure 18.11, both the A7 and the A15 can execute at four distinct operating

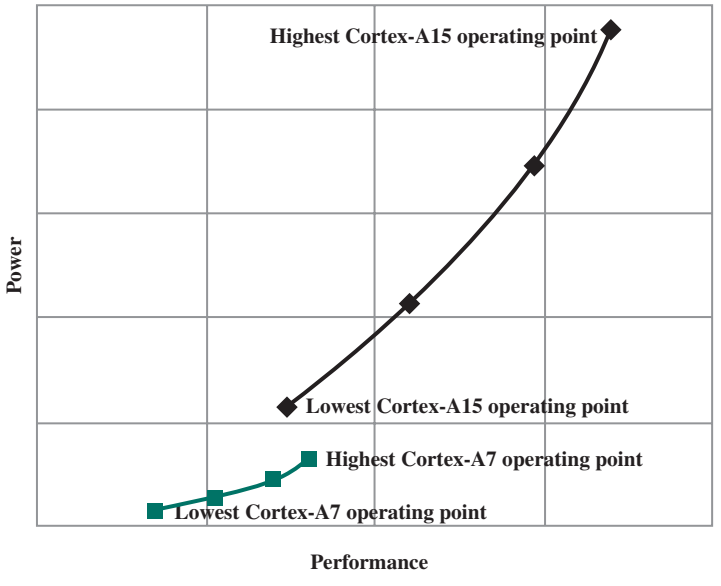


Figure 18.11 Cortex-A7 and A15 Performance Comparison

points. The DVFS software can effectively dial in to one of the operating points on the curve, setting a specific CPU clock frequency and voltage level.

These operating points affect the voltage and frequency of a single CPU cluster; however, in a big.Little system there are two CPU clusters with independent voltage and frequency domains. This allows the big cluster to act as a logical extension of the DVFS operating points provided by the Little processor cluster. In a big.Little system under a migration mode of control, when Cortex-A7 is executing, the DVFS driver can tune the performance of the CPU cluster to higher levels. Once Cortex-A7 is at its highest operating point, if more performance is required, a task migration can be invoked that picks up the OS and applications and moves them to the Cortex-A15. In today's smartphone SoCs, DVFS drivers like cpu_freq sample the OS performance at regular and frequent intervals, and the DVFS governor decides whether to shift to a higher or lower operating point or remain at the current operating point.

The migration model is simple but requires that one of the CPUs in each pair is always idle. The MP model allows any mixture of A15 and A7 cores to be powered on and executing simultaneously. Whether a big processor needs to be powered on is determined by performance requirements of tasks currently executing. If there are demanding tasks, then a big processor can be powered on to execute them. Low demand tasks can execute on a Little processor. Finally, any processors that are not being used can be powered down. This ensures that cores, big or Little, are only active when they are needed, and that the appropriate core is used to execute any given workload.

The MP model is somewhat more complicated to implement but is also more efficient of resources. It assigns tasks appropriately and enables more cores to be running simultaneously when the demand warrants it.

Cache Coherence and the MOESI Model

Typically, a heterogeneous multicore processor will feature dedicated L2 cache assigned to the different processor types. We see that in the general depiction of a CPU/GPU scheme of Figure 18.7. Because the CPU and GPU are engaged in quite different tasks, it makes sense that each has its own L2 cache, shared among the similar CPUs. We also see this in the big.Little architecture (Figure 18.9), in which the A7 cores share an L2 cache and the A15 cores share a separate L2 cache.

When multiple caches exist, there is a need for a cache-coherence scheme to avoid access to invalid data. Cache coherency may be addressed with software-based techniques. In the case where the cache contains stale data, the cached copy may be invalidated and reread from memory when needed again. When memory contains stale data due to a write-back cache containing dirty data, the cache may be cleaned by forcing write back to memory. Any other cached copies that may exist in other caches must be invalidated. This software burden consumes too many resources in a SoC chip, leading to the use of hardware cache-coherent implementations, especially in heterogeneous multicore processors.

As described in Chapter 17, there are two main approaches to hardware-implemented cache coherence: directory protocols and snoopy protocols. ARM has developed a hardware coherence capability called ACE (**A**dvanced Extensible

Interface **C**oherence **E**xtensions) that can be configured to implement either directory or snoopy approach, or even a combination. ACE has been designed to support a wide range of coherent masters with differing capabilities. ACE supports coherency between dissimilar processors such as the Cortex-A15 and Cortex-A7 processors, enabling ARM big.Little technology. It supports I/O coherency for un-cached masters, supports masters with differing cache line sizes, differing internal cache state models, and masters with write-back or write-through caches. As another example, ACE is implemented in the memory subsystem memory controller (MSMC) in the TI SoC chip of Figure 18.8. MSMC supports hardware cache coherence between the ARM CorePac L1/L2 caches and EDMA/IO peripherals for shared SRAM and DDR spaces. This feature allows the sharing of MSMC SRAM and DDR data spaces by these masters on the chip, without having to use explicit software cache maintenance techniques.

ACE makes use of a five-state cache model. In each cache, each line is either Valid or Invalid. If a line is Valid, it can be in one of four states, defined by two dimensions. A line may contain data that are Shared or Unique. A Shared line contains data from a region of external (main) memory that is potentially sharable. A Unique line contains data from a region of memory that is dedicated to the core owning this cache. And the line is either Clean or Dirty, generally meaning either memory contains the latest, most up-to-date data and the cache line is merely a copy of memory, or if it's Dirty then the cache line is the latest, most up-to-date data and it must be written back to memory at some stage. The one exception to the above description is when multiple caches share a line and it's dirty. In this case, all caches must contain the latest data value at all times, but only one may be in the Shared/Dirty state, the others being held in the Shared/Clean state. The Shared/Dirty state is thus used to indicate which cache has responsibility for writing the data back to memory, and Shared/Clean is more accurately described as meaning data is shared but there is no need to write it back to memory.

The ACE states correspond to a cache coherency model with five states, known as MOESI (Figure 18.12). Table 18.2 compares the MOESI model with the MESI model described in Chapter 17.

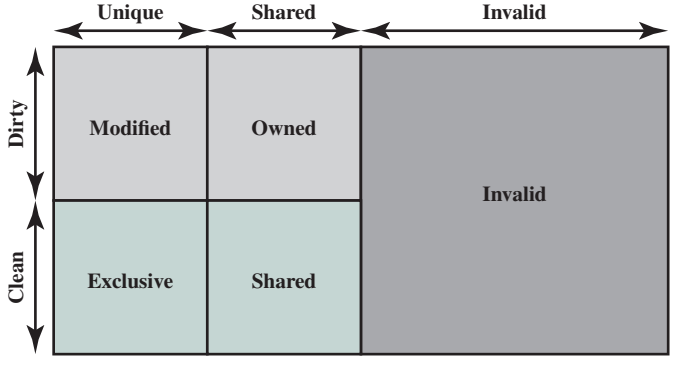


Figure 18.12 ARM ACE Cache Line States

Table 18.2 Comparison of States in Snoop Protocols

(a) MESIM				
	Modified	**Exclusive**	**Shared**	**Invalid**
Clean/Dirty	Dirty	Clean	Clean	N/A
Unique?	Yes	Yes	No	N/A
Can write?	Yes	Yes	No	N/A
Can forward?	Yes	Yes	Yes	N/A
Comments	Must write back to share or replace	Transitions to M on write	Shared implies clean, can forward	Cannot read

(b) MOESI					
	Modified	**Owned**	**Exclusive**	**Shared**	**Invalid**
Clean/Dirty	Dirty	Dirty	Clean	Either	N/A
Unique?	Yes	Yes	Yes	No	N/A
Can write?	Yes	Yes	Yes	No	N/A
Can forward?	Yes	Yes	Yes	No	N/A
Comments	Can share without write back	Must write back to transition	Transitions to M on write	Shared, can be dirty or clean	Cannot read

18.5 INTEL CORE i7-990X

Intel has introduced a number of multicore products in recent years. In this section, we look at the Intel Core i7-990X.

The general structure of the Intel Core i7-990X is shown in Figure 18.13. Each core has its own **dedicated L2 cache** and the six cores share a 12-MB **L3 cache**. One mechanism Intel uses to make its caches more effective is prefetching, in which the hardware examines memory access patterns and attempts to fill the caches speculatively with data that's likely to be requested soon.

The Core i7-990X chip supports two forms of external communications to other chips. The **DDR3 memory controller** brings the memory controller for the DDR main memory[1] onto the chip. The interface supports three channels that are 8 bytes wide for a total bus width of 192 bits, for an aggregate data rate of up to 32 GB/s. With the memory controller on the chip, the Front Side Bus is eliminated.

The **QuickPath Interconnect** (QPI) is a cache-coherent, point-to-point link-based electrical interconnect specification for Intel processors and chipsets. It enables high-speed communications among connected processor chips. The QPI link operates at 6.4 GT/s (transfers per second). At 16 bits per transfer, that adds up to 12.8 GB/s, and since QPI links involve dedicated bidirectional pairs, the total bandwidth is 25.6 GB/s. Section 3.5 covers QPI in some detail.

[1]The DDR synchronous RAM memory is discussed in Chapter 5.

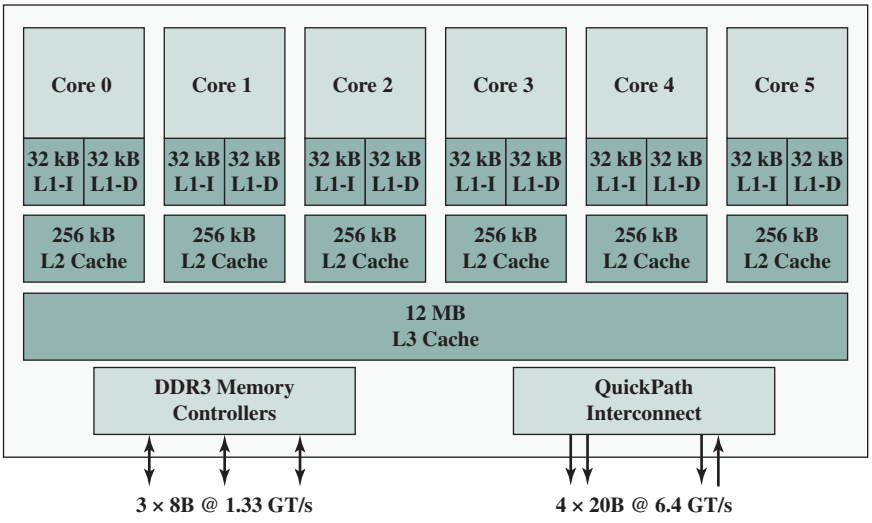


Figure 18.13 Intel Core i7-990X Block Diagram

18.6 ARM CORTEX-A15 MPCORE

We have already seen two examples of heterogeneous multicore processors using ARM cores, in Section 18.4: the big.Little architecture, which uses a combination of ARM Cortex-A7 and Cortex-A15 cores; and the Texas Instruments DSP SoC architecture, which combines Cortex-A15 cores with TI DSP cores. In this section, we introduce the Cortex-A15 MPCore multicore chip, which is a homogeneous multicore processor using multiple A15 cores. The A15 MPCore is a high-performance chip targeted at applications including mobile computing, high-end digital home servers, and wireless infrastructure.

Figure 18.14 presents a block diagram of the Cortex-A15 MPCore. The key elements of the system are as follows:

- **Generic interrupt controller (GIC):** Handles interrupt detection and interrupt prioritization. The GIC distributes interrupts to individual cores.
- **Debug unit and interface:** The debug unit enables an external debug host to: stop program execution; examine and alter process and coprocessor state; examine and alter memory and input/output peripheral state; and restart the processor.
- **Generic timer:** Each core has its own private timer that can generate interrupts.
- **Trace:** Supports performance monitoring and program trace tools.
- **Core:** A single ARM Cortex-15 core.
- **L1 cache:** Each core has its own dedicated L1 data cache and L1 instruction cache.
- **L2 cache:** The shared L2 memory system services L1 instruction and data cache misses from each core.
- **Snoop control unit (SCU):** Responsible for maintaining L1/L2 cache coherency.

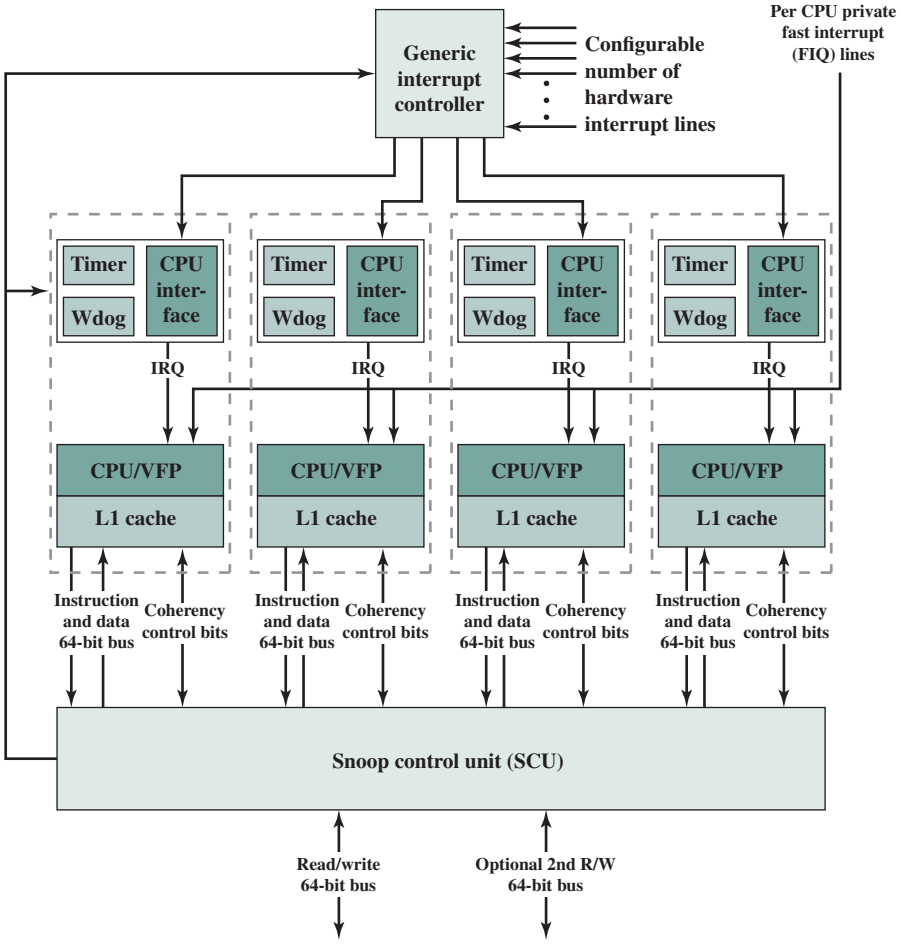


Figure 18.14 ARM Cortex-A15 MPCore Chip Block Diagram

Interrupt Handling

The GIC collates interrupts from a large number of sources. It provides

- Masking of interrupts
- Prioritization of the interrupts
- Distribution of the interrupts to the target A15 cores
- Tracking the status of interrupts
- Generation of interrupts by software

The GIC is a single functional unit that is placed in the system alongside A15 cores. This enables the number of interrupts supported in the system to be independent of the A15 core design. The GIC is memory mapped; that is, control registers for the GIC are defined relative to a main memory base address. The GIC is accessed by the A15 cores using a private interface through the SCU.

The GIC is designed to satisfy two functional requirements:

- Provide a means of routing an interrupt request to a single CPU or multiple CPUs, as required.
- Provide a means of interprocessor communication so that a thread on one CPU can cause activity by a thread on another CPU.

As an example that makes use of both requirements, consider a multithreaded application that has threads running on multiple processors. Suppose the application allocates some virtual memory. To maintain consistency, the operating system must update memory translation tables on all processors. The OS could update the tables on the processor where the virtual memory allocation took place, and then issue an interrupt to all the other processors running this application. The other processors could then use this interrupt's ID to determine that they need to update their memory translation tables.

The GIC can route an interrupt to one or more CPUs in the following three ways:

- An interrupt can be directed to a specific processor only.
- An interrupt can be directed to a defined group of processors. The MPCore views the first processor to accept the interrupt, typically the least loaded, as being best positioned to handle the interrupt.
- An interrupt can be directed to all processors.

From the point of view of software running on a particular CPU, the OS can generate an interrupt to all but self, to self, or to specific other CPUs. For communication between threads running on different CPUs, the interrupt mechanism is typically combined with shared memory for message passing. Thus, when a thread is interrupted by an interprocessor communication interrupt, it reads from the appropriate block of shared memory to retrieve a message from the thread that triggered the interrupt. A total of 16 interrupt IDs per CPU are available for interprocessor communication.

From the point of view of an A15 core, an interrupt can be:

- **Inactive:** An Inactive interrupt is one that is nonasserted, or which in a multiprocessing environment has been completely processed by that CPU but can still be either Pending or Active in some of the CPUs to which it is targeted, and so might not have been cleared at the interrupt source.
- **Pending:** A Pending interrupt is one that has been asserted, and for which processing has not started on that CPU.
- **Active:** An Active interrupt is one that has been started on that CPU, but processing is not complete. An Active interrupt can be pre-empted when a new interrupt of higher priority interrupts A15 core interrupt processing.

Interrupts come from the following sources:

- **Interprocessor interrupts (IPIs):** Each CPU has private interrupts, ID0-ID15, that can only be triggered by software. The priority of an IPI depends on the receiving CPU, not the sending CPU.

- **Private timer and/or watchdog interrupts:** These use interrupt IDs 29 and 30.
- **Legacy FIQ line:** In legacy IRQ mode, the legacy FIQ pin, on a per CPU basis, bypasses the Interrupt Distributor logic and directly drives interrupt requests into the CPU.
- **Hardware interrupts:** Hardware interrupts are triggered by programmable events on associated interrupt input lines. CPUs can support up to 224 interrupt input lines. Hardware interrupts start at ID32.

Figure 18.15 is a block diagram of the GIC. The GIC is configurable to support between 0 and 255 hardware interrupt inputs. The GIC maintains a list of interrupts, showing their priority and status. The Interrupt Distributor transmits to each CPU Interface the highest Pending interrupt for that interface. It receives back the information that the interrupt has been acknowledged, and can then change the status of the corresponding interrupt. The CPU Interface also transmits End of Interrupt (EOI) information, which enables the Interrupt Distributor to update the status of this interrupt from Active to Inactive.

Cache Coherency

The MPCore's Snoop Control Unit (SCU) is designed to resolve most of the traditional bottlenecks related to access to shared data and the scalability limitation introduced by coherence traffic.

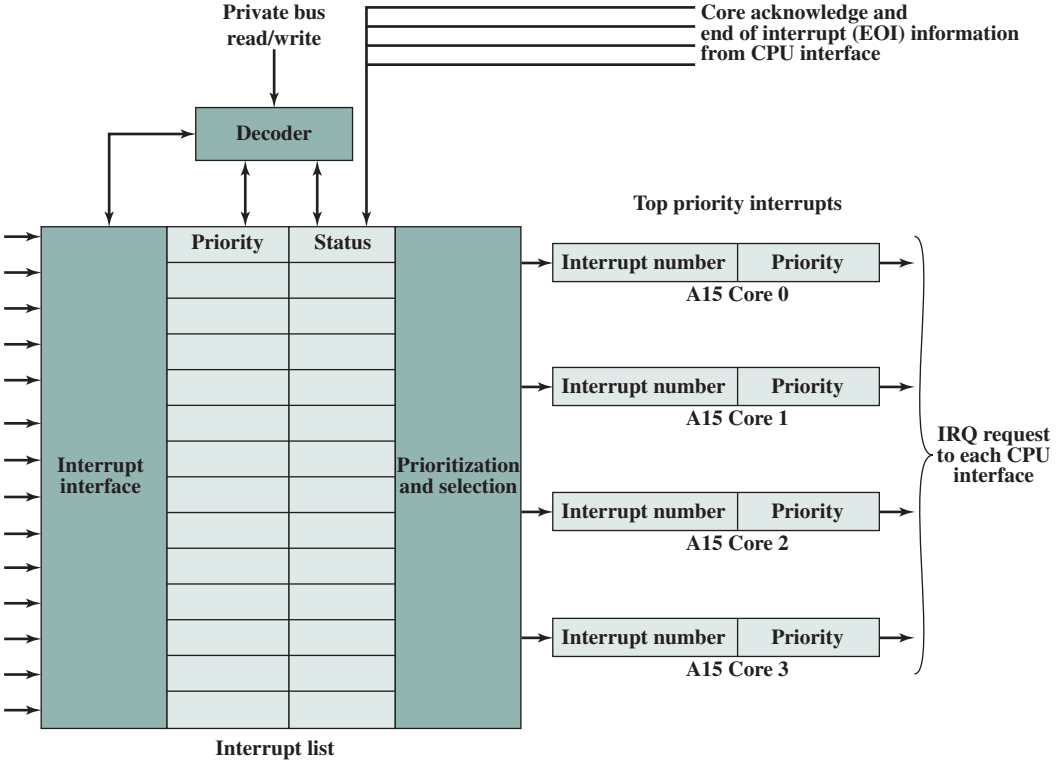


Figure 18.15 Generic Interrupt Controller Block Diagram

L1 CACHE COHERENCY The L1 cache coherency scheme is based on the MESI protocol described in Chapter 17. The SCU monitors operations with shared data to optimize MESI state migration. The SCU introduces three types of optimization: direct data intervention, duplicated tag RAMs, and migratory lines.

Direct data intervention (DDI) enables copying clean data from one CPU L1 data cache to another CPU L1 data cache without accessing external memory. This reduces read after read activity from the Level 1 cache to the Level 2 cache. Thus, a local L1 cache miss is resolved in a remote L1 cache rather than from access to the shared L2 cache.

Recall that main memory location of each line within a cache is identified by a tag for that line. The tags can be implemented as a separate block of RAM of the same length as the number of lines in the cache. In the SCU, **duplicated tag RAMs** are duplicated versions of L1 tag RAMs used by the SCU to check for data availability before sending coherency commands to the relevant CPUs. Coherency commands are sent only to CPUs that must update their coherent data cache. This reduces the power consumption and performance impact from snooping into and manipulating each processor's cache on each memory update. Having tag data available locally lets the SCU limit cache manipulations to processors that have cache lines in common.

The **migratory lines** feature enables moving dirty data from one CPU to another without writing to L2 and reading the data back in from external memory. The operation can be described as follows. In a typical MESI protocol, one processor has a modified line and another processor attempts to read that line, the following actions occur:

1. The line contents are transferred from the modified line to the processor that initiated the read.
2. The line contents are written back to main memory.
3. The line is put in the shared state in both caches.

L2 Cache Coherency

The SCU uses hybrid MESI and MOESI protocols to maintain coherency between the individual L1 data caches and the L2 cache. The L2 memory system contains a snoop tag array that is a duplicate copy of each of the L1 data cache directories. The snoop tag array reduces the amount of snoop traffic between the L2 memory system and the L1 memory system. Any line that resides in the snoop tag array in the Modified/Exclusive state belongs to the L1 memory system. Any access that hits against a line in this state must be serviced by the L1 memory system and passed to the L2 memory system. If the line is invalid or in the shared state in the snoop tag array, then the L2 cache can supply the data. The SCU contains buffers that can handle direct cache-to-cache transfers between cores without reading or writing any data on the ACE. Lines can migrate back and forth without any change to the MOESI state of the line in the L2 cache. Shareable transactions on the ACP are also coherent, so the snoop tag arrays are queried as a result of ACP transactions. For reads where the shareable line resides in one of the L1 data caches in the Modified/Exclusive state, the line is transferred from the L1 memory system to the L2 memory system and passed back on the ACP.

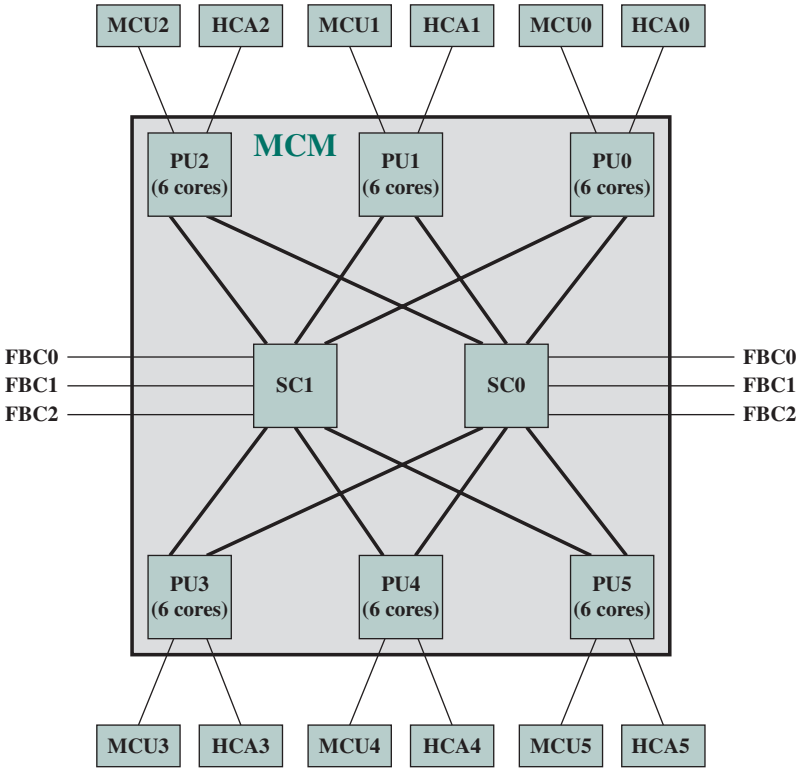
18.7 IBM zENTERPRISE EC12 MAINFRAME

In this section, we look at a mainframe computer organization that uses multicore processor chips. The example we use is the IBM zEnterprise EC12 mainframe computer [SHUM13, DOBO13], which began shipping in late 2010. Section 7.8 provides a general overview of the EC12, together with a discussion of its I/O structure.

Organization

The principal building block of the mainframe is the multichip module (MCM). The MCM is a 103-layer glass ceramic substrate (size 96–96 mm) containing eight chips and 7356 connections. The total number of transistors is over 23 billion. The MCM plugs into a card that is part of the book packaging. The book itself is plugged into the mid-plane system board to provide interconnectivity among the books.

The key components of an MCM are shown in Figure 18.16:



FBC = fabric book connectivity
HCA = host channel adapter
MCM = multichip module
MCU = memory control unit
PU = processor unit
SC = storage control

Figure 18.16 IBM EC12 Processor Node Structure

- **Processor unit (PU):** There are six 5.5-GHz processor PU chips, each containing four processor cores plus three levels of cache. The PUs have external connections to main memory via memory control units and to I/O via host channel adapters. Thus, each MCM includes 24 cores.
- **Storage control (SC):** The two SC chips contain an additional level of cache plus interconnection logic for connecting to three other MCMs.

The microprocessor core features a wide superscalar, out-of-order pipeline that can decode three z/Architecture CISC instructions per clock cycle (< 0.18 ns) and execute up to seven operations per cycle. The instruction execution path is predicted by branch direction and target prediction logic. Each core includes two integer units, two load/store units, one binary floating-point unit, and one decimal floating-point unit.

Cache Structure

The EC12 incorporates a four-level cache structure. We look at each level in turn (Figure 18.17).

Each core has a dedicated 160-kB **L1 cache**, divided into a 96-kB data cache and a 64-kB instruction cache. The L1 cache is designed as a write-through cache to L2, that is, altered data are also stored to the next level of memory. These caches are 8-way set associative.

Each core also has a dedicated 2-MB L2, split equally into 1-MB data cache and 1-MB instruction cache. The L2 caches are write-through to L3, and 8-way set associative.

Each 4-core processor unit chip includes a 24-MB **L3 cache** shared by all six cores. Because L1 and L2 caches are write-through, the L3 cache must process every

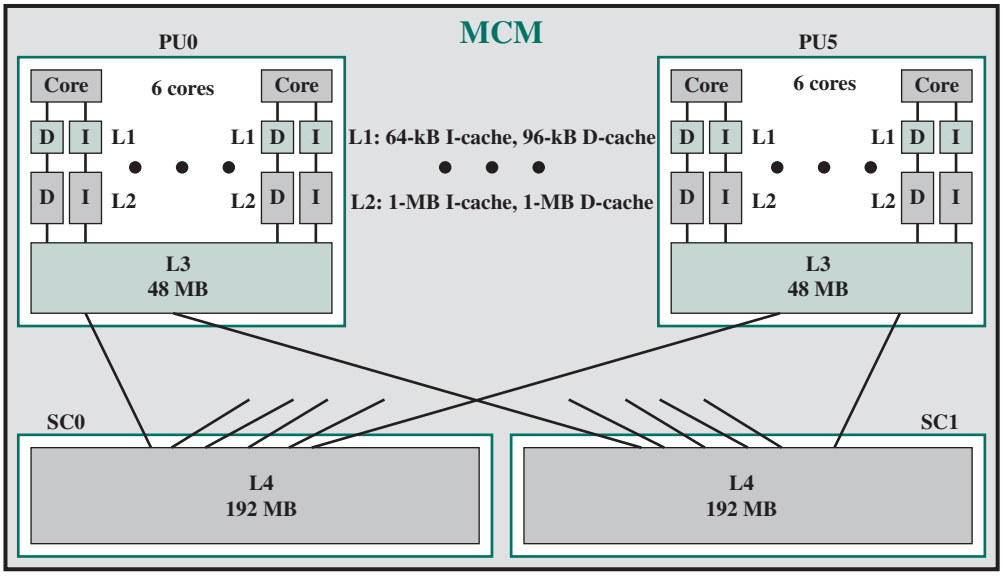


Figure 18.17 IBM EC12 Cache Hierarchy

store generated by the six cores on its chip. This feature maintains data availability during a core failure. The L3 cache is 12-way set associative. The EC12 implements embedded DRAM (eDRAM) as L3 cache memory on the chip. While this eDRAM memory is slower than static RAM (SRAM) normally used to implement cache memory, you can put a lot more of it onto a given area. For many workloads, having more memory closer to the core is more important than having fast memory.

Finally, all 6 PUs on an MCM share a 160-MB **L4 cache**, which is split into one 92-MB cache on each SC chip. The principal motivation for incorporating a level 4 cache is that the very high clock speed of the core processors results in a significant mismatch with main memory speed. The fourth cache layer is needed to keep the cores running efficiently. The large shared L3 and L4 caches are suited to transaction-processing workloads exhibiting a high degree of data sharing and task swapping. The L4 cache is 24-way set associative. The SC chip, which houses the L4 cache, also acts as a cross-point switch for L4-to-L4 traffic to up to three remote books[2] by three bidirectional data buses. The L4 cache is the coherence manager, meaning that all memory fetches must be in the L4 cache before that data can be used by the processor.

All four caches use a line size of 256 bytes.

The EC12 is an interesting study in design trade-offs and the difficulty in exploiting the increasingly powerful processors available with current technology. The large L4 cache is intended to drive the need for access to main memory down to the bare minimum. However, the distance to the off-chip L4 cache costs a number of instruction cycles. Thus, the on-chip area devoted to cache is as large as possible, even to the point of having fewer cores than possible on the chip. The L1 caches are small, to minimize distance from the core and ensure that access can occur in one cycle. Each L2 cache is dedicated to a single core, in an attempt to maximize the amount of cached data that can be accessed without resort to a shared cache. The L3 cache is shared by all four cores on a chip and is as large as possible, to minimize the need to go to the L4 cache.

Because all of the books of the zEnterprise 196 share the workload, the four L4 caches on the four books form a single pool of L4 cache memory. Thus, access to L4 means not only going off-chip but perhaps off-book, further increasing access delay. This means relatively large distances exist between the higher-level caches in the processors and the L4 cache content. Still, accessing L4 cache data on another book is faster than accessing DRAM on the other book, which is why the L4 caches work this way.

To overcome the delays that are inherent to the book design and to save cycles to access the off-book L4 content, the designers try to keep instructions and data as close to the cores as possible by directing as much work as possible of a given logical partition workload to the cores located in the same book as the L4 cache. This is achieved by having the system resource manager/scheduler and the z/OS dispatcher work together to keep as much work as possible within the boundaries of as few cores and L4 cache space (which is best within a book boundary) as can be achieved without affecting throughput and response times. Preventing the resource manager/scheduler and the dispatcher from assigning workloads to processors where they might run less efficiently contributes to overcoming latency in a high-frequency processor design such as the EC12.

---

[2]Recall from Chapter 7 that a EC12 book consists of an MCM, memory cards, and I/O cage connections.

18.8 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

Amdahl's law	homogenous multicore organization	Pollack's rule
chip multiprocessor	hybrid threading	simultaneous multithreading (SMT)
coarse-grained threading	MOESI protocol	superscalar
fine-grained threading	multicore processor	threading granularity
heterogeneous multicore organization	pipelining	

Review Questions

18.1 Summarize the differences among simple instruction pipelining, superscalar, and simultaneous multithreading.

18.2 Give several reasons for the choice by designers to move to a multicore organization rather than increase parallelism within a single processor.

18.3 Why is there a trend toward giving an increasing fraction of chip area to cache memory?

18.4 List some examples of applications that benefit directly from the ability to scale throughput with the number of cores.

18.5 At a top level, what are the main design variables in a multicore organization?

18.6 List some advantages of a shared L2 cache among cores compared to separate dedicated L2 caches for each core.

Problems

18.1 Consider the following problem. A designer has a chip available and must decide what fraction of the chip will be devoted to cache memory (L1, L2, L3). The remainder of the chip will be devoted to one or more complex superscalar and/or SMT cores. Define the following parameters:

- n = maximum number of cores that can be contained on the chip.
- k = actual number of cores implemented ($1 \leq k \leq n$, where $r = n/k$ is an integer).
- $perf(r)$ = sequential performance gain by using the resources equivalent to r cores to form a single processor, where $perf(1) = 1$.
- f = fraction of software that is parallelizable across multiple cores.

Thus, if we construct a chip with n cores, we expect each core to provide sequential performance of 1 and for the n cores to be able to exploit parallelism up to a degree of n parallel threads. Similarly, if the chip has k cores, then each core should exhibit a performance of $perf(r)$ and the chip is able to exploit parallelism up to a degree of k parallel threads. We can modify Amdhal's law (Equation 18.1) to reflect this situation as follows:

$$\text{Speedup} = \frac{1}{\frac{1 - -f}{perf(r)} + \frac{f \times r}{perf(r) \times n}}$$

a. Justify this modification of Amdahl's law.
b. Using Pollack's rule, we set $perf(r) = \sqrt{r}$. Let $n = 16$. We want to plot speedup as a function of r for $f = 0.5$; $f = 0.9$; $f = 0.975$; $f = 0.99$; $f = 0.999$. The results are available in a document at this book's Premium Content site (multicore-performance.pdf). What conclusions can you draw?
c. Repeat part (b) for $n = 256$.

18.2 The technical reference manual for the Cortex-A15 says that the GIC is memory mapped. That is, the core processors use memory mapped I/O to communicate with the GIC. Recall from Chapter 7 that with memory mapped I/O, there is a single address space for memory locations and I/O devices. The processor treats the status and data registers of I/O modules as memory locations and uses the same machine instructions to access both memory and I/O devices. Based on this information, what path through the block diagram of Figure 18.15 is used for the core processors to communicate with the GIC?

18.3 In this question we analyze the performance of the following C program on a multithreaded architecture. You should assume that arrays A, B, and C do not overlap in memory.

```
for (i=0; i<328; i++) {
          A[i] = A[i]*B[i];
          C[i] = C[i]+A[i];
          }
```

- Our machine is a single-issue, in-order processor. It switches to a different thread every cycle using fixed round robin scheduling. Each of the N threads executes one instruction every N cycles. We allocate the code to the threads such that every thread executes every Nth iteration of the original C code.
- Integer instructions take 1 cycle to execute, floating-point instructions take 4 cycles and memory instructions take 3 cycles. All execution units are fully pipelined. If an instruction cannot issue because its data is not yet available, it inserts a bubble into the pipeline, and retries after I cycles.
- Below is our program in assembly code for this machine for a single thread executing the entire loop.

```
loop: ld f1, 0 (r1)       ;f1 = A[i]
      ld f2, 0 (r2)       ;f2 = B[i]
      fmul f4, f2, f1     ;f4 = f1*f2
      st f4 0(r1)         ;A[i] = f4
      ld f3, 0(r3)        ;f3 = C[i]
      fadd f5, f4, f3     ;f5 = f4 + f3
      st f5 0(r3)         ;C[i] = f5
      add r1, r1, 4       ;i++
      add r2, r2, 4
      add r3, r3, 4
      add r4, r4, -1
      bnez r4, loop       ;loop
```

a. We allocate the assembly code of the loop to N threads such that every thread executes every Nth iteration of the original loop. Write the assembly code that one of the N threads would execute on this multithreaded machine.
b. What is the minimum number of threads this machine needs to remain fully utilized issuing an instruction every cycle for our program?
c. Could we reach peak performance running this program using fewer threads by rearranging the instructions? Explain briefly.
d. What will be the peak performance in flops/cycle for this program?

18.4 For the MOESI protocol, consider any pair of caches. Use the following matrix to indicate which states are permitted for a given cache line; use X for forbidden and checkmark for permitted.

	M	O	E	S	I
M					
O					
E					
S					
I					

18.5 Draw a state transition diagram, including labels on the transitions, for the MOESI protocol, similar to Figure 17.6.

18.6 In directory cache coherence protocols, such as those based on MESI or MOESI, a silent transition is one in which a cache line transitions from one state to another without reporting this change to the central controller.

a. For each state in the MESI protocol, indicate to which target states, if any, a silent transition is possible.

b. Repeat for MOESI.

CHAPTER 19

General-Purpose Graphic Processing Units

Contributed by
Peter Zeno
Ph.D. Candidate, University of Bridgeport

19.1 CUDA Basics

19.2 GPU versus CPU
Basic Differences between CPU and GPU Architectures
Performance and Performance per Watt Comparison

19.3 GPU Architecture Overview
Baseline GPU Architecture
Full Chip Layout
Streaming Multiprocessor Architecture Details
Importance of Knowing and Programming to Your Memory Types

19.4 Intel's Gen8 GPU

19.5 When to Use a GPU as a Coprocessor

19.6 Key Terms and Review Questions

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Present an overview of CUDA.
- Understand the difference between a GPU and a CPU.
- Describe the basic elements of a typical GPU architecture.
- Discuss when to use a GPU as a coprocessor.

The **graphics processor unit (GPU)** is designed specifically to be optimized for fast three-dimensional (3D) graphics rendering and video processing. GPUs can be found in almost all of today's workstations, laptops, tablets, and smartphones [OWEN08]. The GPU comes in many sizes. The larger units have several hundred to thousands of parallel processor cores on a single integrated circuit (IC). These can be found as separate coprocessor cards, usually PCIe-based, in workstations, gaming systems, and even supercomputers [SLAV12]. The smallest GPUs are found in embedded systems, such as tablets and smartphones, where the GPU is composed of only a single-digit number of cores, and are typically combined with a number of conventional cores, referred to as **central processing units (CPUs)** on the same silicon IC.

Over the past several years, the GPU has found its way into massively parallel programming environments for a wide range of applications, such as bioinformatics, molecular dynamics, oil and gas exploration, computational finance, signal and audio processing, statistical modeling, computer vision, and medical imaging. This is where the term **general-purpose computing using a GPU (GPGPU)** is derived from. The main reasons for the migration of highly parallelizable applications to the GPU are due to the advent of programmer friendly GPGPU languages, such as NVIDIA's CUDA and the Khronos Group's OpenCL, some slight modifications to the GPU architecture to facilitate general-purpose computing [SAND10] (from here on known as GPGPU architecture), along with the low cost and high performance of GPUs. For example, for about $200, one can purchase a GPU with 960 parallel processor cores for your workstation (e.g., NVIDIA's GeForce GTX 660).

We begin this chapter with an overview of the CUDA model, which is essential for understanding the design and use of GPUs. Next, the chapter contrasts GPUs and CPUs. This is followed by a detailed look at GPU architecture. Then, Intel's GPU is examined. Finally, the chapter discusses when to use a GPU as a coprocessor.

19.1 CUDA BASICS

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they produce. To adequately describe the GPGPU architecture, several CUDA software terms and concepts need to be covered first. This is by no means a comprehensive introduction to the CUDA programming language, particularly since the focus of this chapter and book is on computer architecture. However, it

is difficult to describe the hardware portion of the GPGPU system without first laying the foundation with CUDA software terminology and its programming framework. These concepts will carry over into the GPU/GPGPU architecture domain.

CUDA C is a C/C++ based language. A CUDA program can be divided into three general sections: (1) code to be run on the host (CPU); (2) code to be run on the device (GPU); and (3) the code related to the transfer of data between the host and the device. The code to be run on the host is of course serial code that can't, or isn't worth, parallelizing. The data-parallel code to be run on the GPU is called a **kernel**, while a **thread** is a single instance of this kernel function. The kernel typically will have few to no branching statements. Branching statements in the kernel result in serial execution of the threads in the GPU hardware. More about this will be covered in Section 19.3.

The programmer defines the number of threads launched when the kernel function is called. The total number of threads defined is typically in the thousands to maximize the utilization of the **GPU processor cores** (also known as **CUDA cores**), as well as maximize the available speedup. Additionally, the programmer specifies how these threads are to be bundled. To be more specific, threads are uniformly bundled in **blocks**, and the number of blocks (also known as **thread blocks**) per kernel launch is called a **grid** (see Figure 19.1). Table 19.1 gives a summary of the CUDA terms just defined.

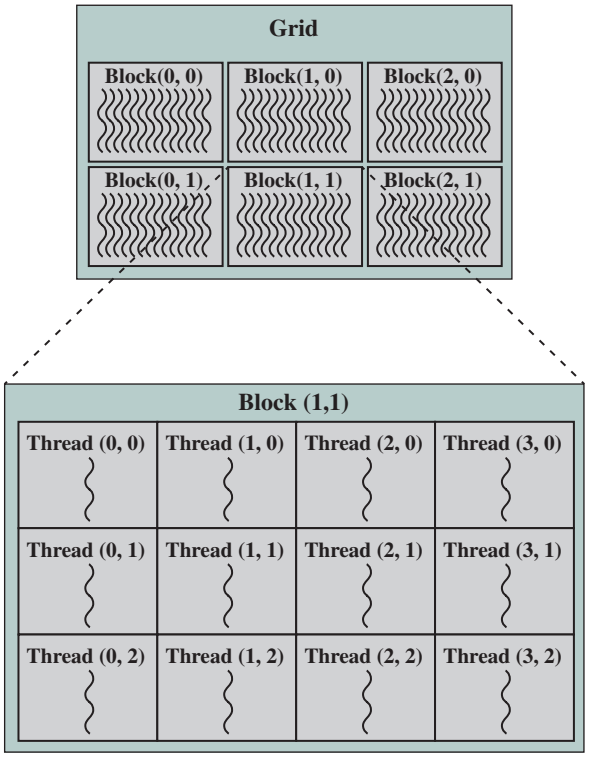


Figure 19.1 Relationship among Threads, Blocks, and a Grid

Table 19.1 CUDA Terms to GPU's Hardware Components Equivalence Mapping

CUDA Term	Definition	Equivalent GPU Hardware Component
Kernel	Parallel code in the form of a function to be run on GPU	Not applicable
Thread	An instance of the kernel on the GPU	GPU/CUDA processor core
Block	A group of threads assigned to a particular SM	CUDA multiprocessor (SM)
Grid	The GPU	GPU

Figure 19.1 illustrates a two-dimensional **grid** of two-dimensional thread blocks. Both the grid and block dimensions can be either one, two, or three dimensions. They need not have the same dimensions. For example, the grid could be set to one dimension, and the thread block could be set to three dimensions. However, as we will see shortly, this configuration can't fully utilize the GPU processors, because a block is assigned to only one of the several GPU **streaming multiprocessors (SMs)**. A block is never split between SMs. Thus, all but one set of GPU processor cores will be idle, while one SM is bearing the full processing load. Additionally, there is a maximum number of threads that an SM will accept. If this number is surpassed, then the code won't compile. Therefore, it is up to the programmer to use the specification data of the GPU to be used, and distribute the load as uniformly as possible. At minimum, the number of thread blocks launched should be no less than the number of SMs on the GPU. However, finding the optimum configuration can be a very time consuming and daunting process.

19.2 GPU VERSUS CPU

This section compares the complementary architectures of the GPU and the CPU. Because the GPU and CPU are orthogonally optimized to one another, their combination into a heterogeneous GPGPU system provides superior cost and performance gains for certain applications, compared to a pure CPU approach.

Basic Differences between CPU and GPU Architectures

Because the GPU and the CPU are designed and optimized for two significantly different types of applications, their architectures differ significantly. This can be seen by comparing the relative amount of die area (transistor count) that is dedicated to cache, control logic, and processing logic for the two types of processor technologies (see Figure 19.2). In the CPU, as discussed in Chapter 18, the control logic and cache memory make up the majority of the CPU's real estate. This is as expected for an architecture which is tuned to process sequential code as quickly as possible. On the other hand, a GPU uses a massively parallel SIMD (single instruction multiple data) architecture to perform mainly mathematical operations. As such, a GPU doesn't require the same complex capabilities of the CPU's control logic (i.e., out of order execution, branch prediction, data hazards, etc.). Nor does it require large amounts of cache memory. GPUs simply run the same thread of code on large amounts of data

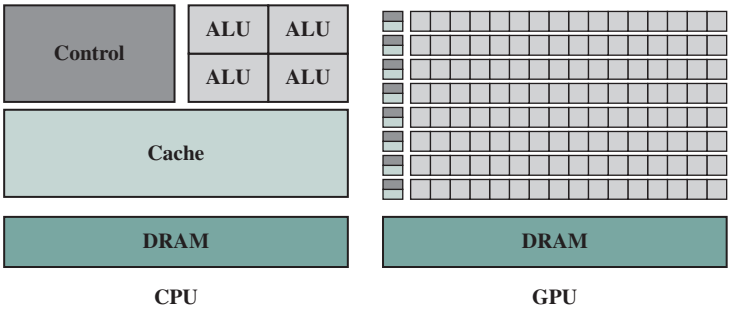


Figure 19.2 CPU versus GPU Silicon Area/Transistor Dedication

and are able to hide memory latency by managing the execution of more threads than available processor cores.

Performance and Performance per Watt Comparison

The video game market has driven the need for ever-increasing real-time graphics realism. This translates into more parallel GPU processor cores with greater floating-point capabilities. As a result, the GPU is designed to maximize the number of floating-point operations per second (FLOPs) it can perform. Additionally, newer NVIDIA architectures, such as the Kepler and Maxwell architectures, have focused on increasing the performance per watt ratio (FLOPs/watt) over previous GPU architectures by decreasing the power required by each GPU processor core. This was accomplished with Kepler by decreasing its processor cores' clock, while increasing the number of on-chip transistors (following Moore's Law) allowing for a positive net gain of 3x the performance per watt over the Fermi architecture. Additionally, the Maxwell architecture has improved execution efficiency. This trend of increasing FLOPs that a GPU can perform versus a multicore CPU has diverged at an exponential rate (see Figure 19.3 [NVID14]), thus creating a large performance gap. Similar can be said about the performance per watt gap between these two different processing technologies.

19.3 GPU ARCHITECTURE OVERVIEW

The historical evolution of the GPU architecture can be divided up into three major phases or eras. The first phase would cover the early days of the GPU architecture (early 1980s to late 1990s), where the GPU was composed of fixed, nonprogrammable, specialized processing stages (e.g., vertex, raster, shader, etc.). Additionally, the continued technology advancements during this period, allowing for a dramatic decrease in the size and cost of a graphics system, in turn brought graphics processors to the PC in the mid- to late-1990s. The second phase would cover the iterative modification of the resulting Phase I GPU architecture from a fixed, specialized, hardware pipeline to a fully programmable processor (approximately during the early to mid-2000s). The general, final modification, introduced by NVIDIA in 2006, facilitated the use of its new GPGPU language, CUDA. The third phase picks up where the second one leaves off and covers how the GPU/GPGPU architecture makes an excellent and affordable highly parallelized SIMD coprocessor for

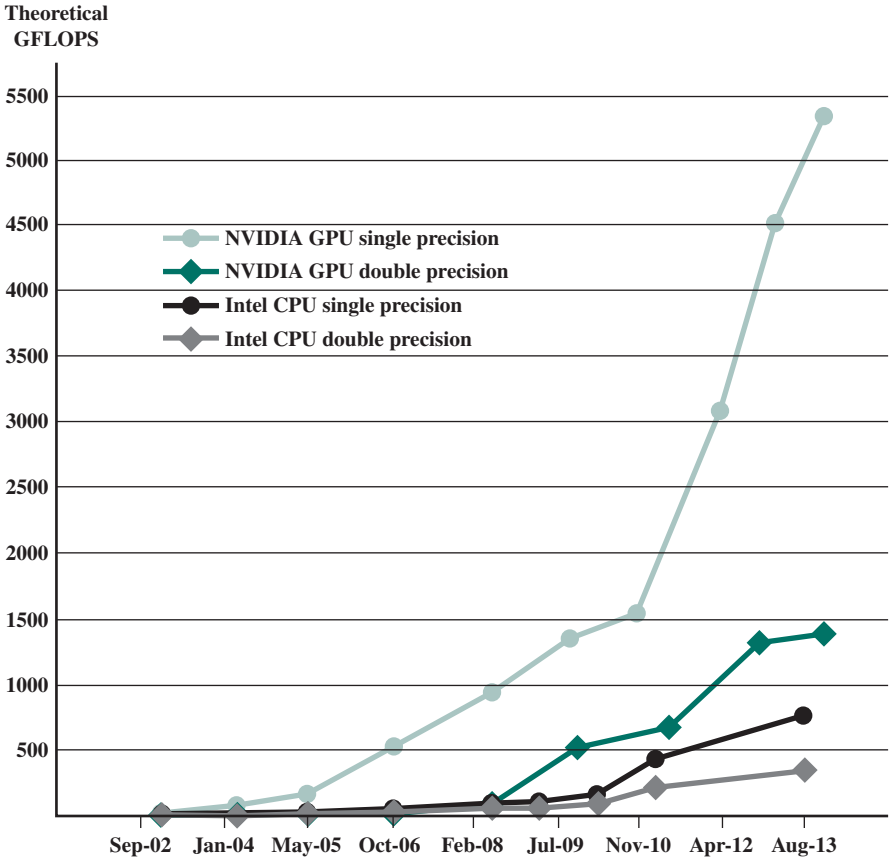


Figure 19.3 Floating-Point Operations per Second for CPU and GPU

accelerating the run times of some nongraphics-related programs, along with how a GPGPU language (CUDA in this case) maps to this architecture. The focus of this chapter follows this third phase or era of the GPU.

The first NVIDIA GPU with added GPGPU support hardware was the GeForce 8800 GTX. To enable the GPU to be used by programmers for general-purpose parallel computing applications, a true cache hierarchy and a user-addressable shared memory was added. Additionally, arrays of the programmable GPU processor cores are equally divided up into scalable SMs. The benefit of such an architecture is the scalability of GPU processor cores, as well as SMs in new generations or different models of GPUs without requiring modification to the CUDA programming language.

Baseline GPU Architecture

As previously mentioned, NVIDIA has progressed through several generations of GPU processing technologies (i.e., Tesla, Fermi, Kepler, and Maxwell), each of which has a small to moderate difference in its microarchitecture over its predecessor. The naming convention for the SM has been slightly modified for the newer generations

of GPU technologies, such as SMX for Kepler and SMM for Maxwell. This helps signify a relatively significant change to the SM architecture from its predecessor (it also helps with the new product's promotional marketing!). With that being said, from a CUDA programming perspective, all of these processing technologies still have identical top-level architectures.

For the remainder of this chapter, we will use NVIDIA's Fermi architecture as the example baseline architecture. The Fermi architecture was chosen due to its fairly representative GPU architecture and lower CUDA core/SM count, which simplifies the mapping between the GPU hardware and the CUDA software. This example architecture is composed of 16 SMs, where each SM contains a group of 32 CUDA cores. Therefore, the Fermi GPU has a total of 16 SMs × 32 CUDA cores/SM, or 512 CUDA cores.

Full Chip Layout

Figure 19.4 illustrates the general layout of the NVIDIA Fermi architecture GPU. As can be seen in this figure, the L2 cache is centrally located to the 16 SMs (8 SMs above and below). Each SM is represented by the 2 adjacent columns and 16 rows of rectangles (GPU processor cores), along with a column of 16 load/store units and a column of 4 special function units (SFUs). A more detailed illustration of the SM module is shown in Figure 19.5 [NIVD09]. The rectangles at the head and foot of the SMs in Figure 19.4 are where the registers and L1/shared memory are located. Each of the six DRAM I/O interfaces has a 64-bit memory interface (the DRAM interface circuitry is show in dark blue rectangles on the outermost left and right sides). Thus, collectively, there is a 384-bit interface to the GPU's GDDR5 (graphic double data

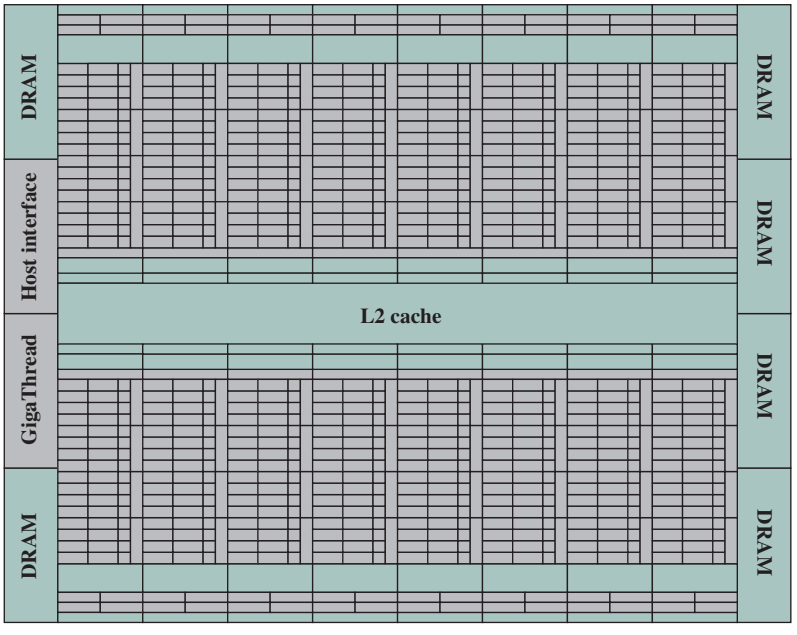


Figure 19.4 NVIDIA Fermi Architecture

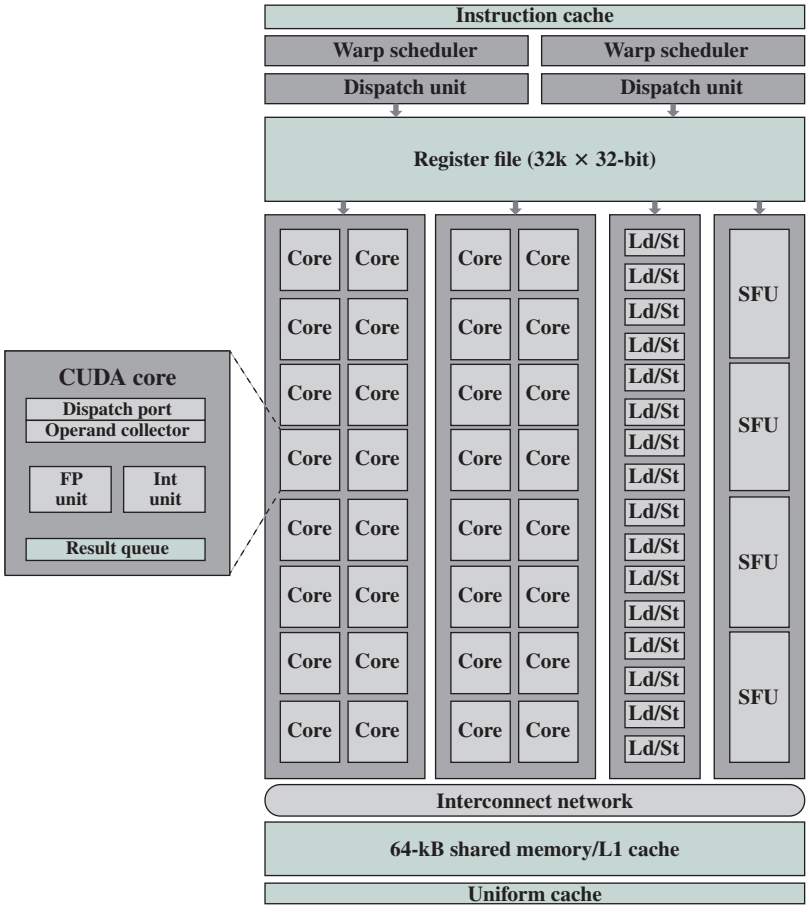


Figure 19.5 Single SM Architecture

rate, a DDR memory designed specifically for graphic processing) DRAM, allowing for support of up to a total of 6 GB of SM off-chip memory (i.e., global, constant, texture, and local). More specifics about these different memory types will be discussed in the next section. Also, illustrated in Figure 19.4 is the host interface, which can be found on the left-hand side of the GPU layout diagram. The host interface allows for PCIe connectivity between the GPU and the CPU. Lastly, the GigaThread global scheduler, in orange and located next to the host interface, is responsible for the distribution of thread blocks to all of the SM's warp schedulers (see Figure 19.5).

Streaming Multiprocessor Architecture Details

The right-hand side of Figure 19.5 breaks down the NVIDIA Fermi architecture into its basic components for a single SM. These components are

- GPU processor cores (total of 32 CUDA cores)
- Warp scheduler and dispatch port

- Sixteen load/store units
- Four SFUs
- 32k × 32-bit registers
- Shared memory and L1 cache (64 kB in total)

DUAL WARP SCHEDULER As covered previously, the GigaThread global scheduler unit on the GPU chip distributes the thread blocks to the SMs. The dual warp scheduler will then break up each thread block it is processing into **warps**, where a warp is a bundle of 32 threads that start at the same starting address and their thread IDs are consecutive. Once a warp is issued, each thread will have its own instruction address counter and register set. This allows for independent branching and execution of each thread in the SM.

The GPU is most efficient when it is processing as many warps as possible to keep the CUDA cores maximally utilized. As illustrated in Figure 19.6, maximum SM hardware utilization will occur when the dual warp schedulers and instruction dispatch units are able to issue two warps every two clock cycles (Fermi architecture). As explained next, structural hazards are the main source of an SM falling short of achieving this maximum processing rate, while off-chip memory access latency can be more easily hidden.

Each divided column of 16 CUDA cores (× 2), 16 load/store units, and 4 SFUs (see Figure 19.5) is eligible to be assigned half a warp (16 threads) to process from each of the two warp scheduler/dispatch units per clock cycle, given that the component column isn't experiencing a structural hazard. Structural hazards are caused by limited SFUs, double-precision multiplication, and branching. However, the warp schedulers have a built-in scoreboard to track warps that are available for execution, as well as structural hazards. This allows for the SM to both work around structural hazards and help hide off-chip memory access latency as optimally as possible.

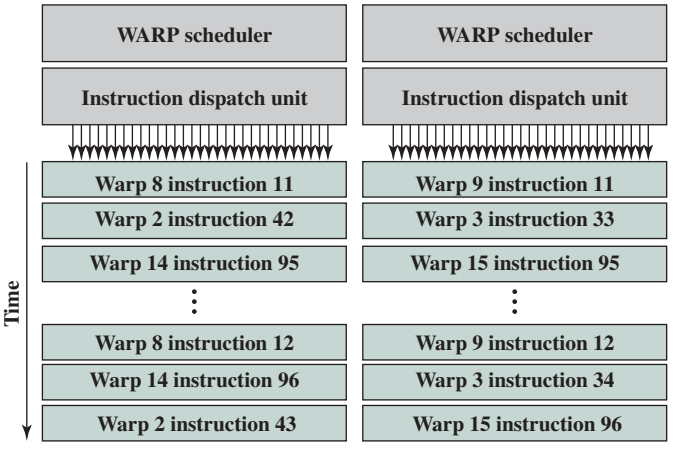


Figure 19.6 Dual Warp Schedulers and Instruction Dispatch Units Run Example

Therefore, it is important for the programmer to set the thread block size greater than the total number of CUDA cores in an SM, but less than the maximum allowable threads per block, and to make sure the thread block size (in the *x* and/or *y* dimensions) is a multiple of 32 (warp size) to achieve near-optimal utilization of the SMs.

CUDA CORES As mentioned in the CUDA Basics section, the NVIDIA GPU processor cores are also known as CUDA cores (see Figure 19.5). Also defined earlier, and as can be seen in Figure 19.4, there are a total of 32 CUDA cores dedicated to each SM in the Fermi architecture. Each CUDA core has two separate pipelines or data paths: an integer (INT) unit pipeline and a floating-point (FP) unit pipeline (see Figure 19.5). Only one of these data paths can be used during a single clock period. The INT unit is capable of 32-bit, 64-bit, and extended precision for integer and logic/bitwise operations. The FP unit can perform a single-precision FP operation, while a double-precision FP operation requires two CUDA cores. Therefore, threads that perform only double-precision FP operations will take twice as long to run compared to a single-precision FP thread. This performance impact of double-precision FP arithmetic is addressed in the Kepler architecture by the inclusion of dedicated double-precision units in each SM, as well as a majority of single-precision units. Fortunately, the management of thread-level FP single- and double-precision operations is hidden from the CUDA programmer. However, the programmer should be aware of the potential performance impact that can be incurred between using the two precision types based on the GPU used.

The Fermi architecture added an improvement to the CUDA core's FP unit over its predecessors. It upgraded from the IEEE 754-1985 floating-point arithmetic standard to the IEEE 754-2008 standard. This was accomplished by improving on the accuracy of the multiply-add instruction (MAD) with a fused multiply-add (FMA) instruction. The FMA instruction is valid for both single- and double-precision arithmetic. The Fermi architecture performs only a single rounding at the end of an FMA instruction. Not only is the accuracy of the result improved, but also performing an FMA instruction is compressed into a single processor clock cycle. Therefore, 32 single-precision or 16 double-precision FMA operations can occur in a single processor clock cycle per SM.

SPECIAL FUNCTION UNITS Each SM has four SFUs. The SFU performs transcendental operations, such as cosine, sine, reciprocal, and square root, in a single clock cycle. Since there are only 4 SFUs in an SM and 32 parallel threads of a single instruction in a warp, it takes 8 clock cycles to complete a warp that requires the SFUs. However, the CUDA processors, along with the load and store units, can still be utilized at the same time.

LOAD AND STORE UNITS Each of the 16 load and store units of the SM calculates the source and destination addresses for a single thread per clock cycle. The addresses are for the cache or DRAM that the threads wish to write data to, or read data from.

REGISTERS, SHARED MEMORY, AND L1 CACHE As illustrated in Figure 19.5, each SM has its own (on-chip) dedicated set of registers and shared memory/L1 cache block. Details and benefits as to these low latency, on-chip memories are described below.

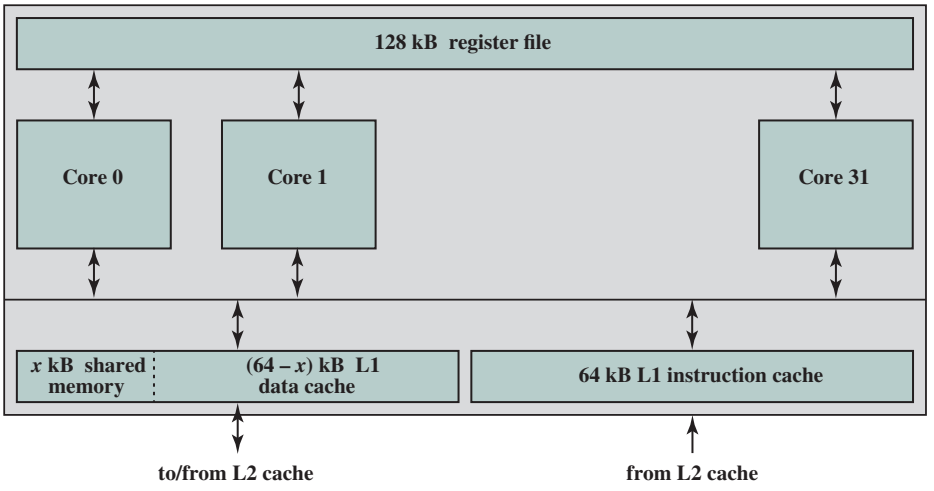
Table 19.2 GPU's Memory Hierarchy Attributes

Memory Type	Relative Access Times	Access Type	Scope	Data Lifetime
Registers	Fastest. On-chip	R/W	Single thread	Thread
Shared	Fast. On-chip	R/W	All threads in a block	Block
Local	100 × to 150 × slower than shared and register. Off-chip	R/W	Single thread	Thread
Global	100 × to 150 × slower than shared and register. Off-chip.	R/W	All threads and host	Application
Constant	100 × to 150 × slower than shared and register. Off-chip	R	All threads and host	Application
Texture	100 × to 150 × slower than shared and register. Off-chip	R	All threads and host	Application

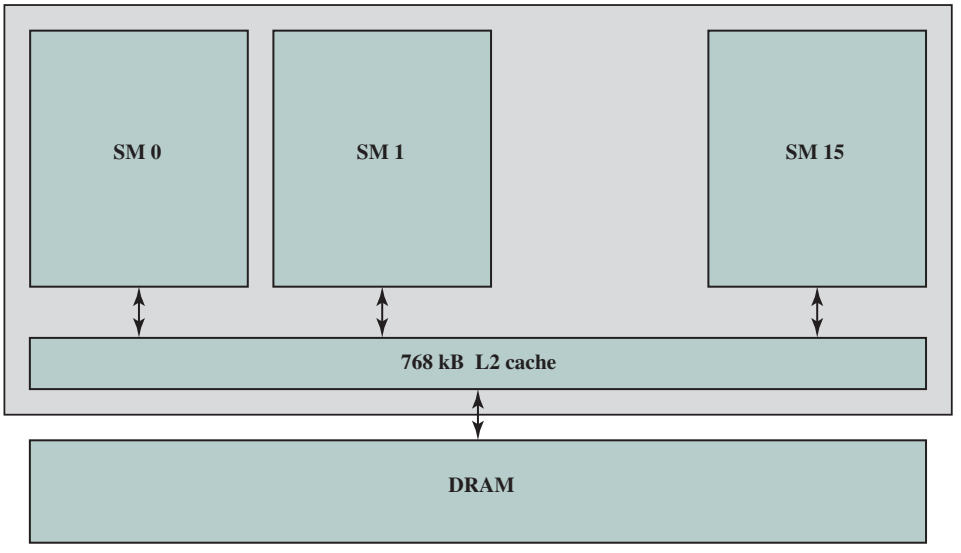
Although the Fermi architecture has an impressive 32k × 32-bit registers per SM, each thread has a maximum of 64 × 32-bit registers allocated to it as defined by CUDA compute capability version 2.x, which is a function of the maximum number of active warps allowed per SM, as well as the number of registers per SM. As shown in Table 19.2, the registers, along with shared memory, have the fastest access times of only several nanoseconds (ns). If there is any temporary register spillage, the data will first get moved to L1 cache before being sent to L2 cache, then long access latency local memory (see Figure 19.7a). The use of L1 cache helps prevent data read/write hazards from occurring. The lifetime of the data in the registers assigned to a thread is therefore only as long as the life of the thread.

The addressable, on-chip shared memory dedicated to the GPU processor cores of an SM is a unique configuration when compared to contemporary multicore microprocessors, such as the CPU. These contemporary architectures, as covered in Chapter 18 and illustrated in Figure 18.6, have a dedicated on-chip L1 cache and a set of registers per core. However, they typically do not have on-chip addressable memory. Instead, dedicated memory management hardware regulates the movement of data between the cache and main memory without control from the programmer. This is significantly different from the GPU architecture (see Figure 19.5).

As discussed at the beginning of this chapter, shared memory was added to the GPU architecture specifically to assist with GPGPU applications. Optimizing the use of shared memory can significantly improve the speedup and performance of a GPGPU application by eliminating unneeded long latency accesses to off-chip memory. Despite the shared memory being small in size for each SM (48 kB at maximum configuration), it has a very low access latency of 100 × to 150 × less than global memory (see Table 19.2). Thus, there are three major ways that shared memory can accelerate the parallel processing tasks: (1) multiple repeated use of shared memory data by all threads of a block (e.g., blocks of data used for matrix–matrix multiplication); (2) select threads of a block (based on specific IDs) are used to transfer data from the global memory to the shared memory, thus redundant



(a) SM memory architecture



(b) Overall memory architecture

Figure 19.7 Fermi Memory Architecture

reads and writes to the same memory locations are removed; and (3) the user can optimize data accesses to global memory by making sure the accesses are coalesced, when possible. All of these points also aid in reducing off-chip memory bandwidth constraint issues. The lifetime of the data in an SM's shared memory is as long as the life of the thread block being processed on it. So, once all of the threads of the block have completed, the data in the SM's shared memory is no longer valid.

Although the use of shared memory will give the optimum run times, in some applications the memory accesses are not known during the programming phase. This is where having more L1 cache available (maximum setting of 48 kB) will give the optimal results. Additionally, the L1 cache helps with aiding register spills,

instead of going straight to local (off-chip) DRAM memory. The two-level cache hierarchy—single L1 cache per SM, and the across chip, SM shared L2 cache—gives the same benefits as those found in conventional multicore microprocessors.

Importance of Knowing and Programming to Your Memory Types

It is important for the programmer to understand the nuances of the various GPU memories, particularly the sizes available for each memory type, their relative access times, and accessibility limitations, to enable correct and efficient code development using CUDA. As one can see from the CUDA Basics section covered at the beginning of the chapter, the SM level memories just covered, and the terminology and parameters listed in Table 19.2, a much different approach is required for GPGPU programming than program development targeted for a CPU, where the specific data storage hardware used (other than file I/O) is hidden from the programmer.

For example, with the GPU architecture, each thread assigned to a CUDA core has its own set of registers, such that one thread cannot access another thread's registers, whether in the same SM or not. The only way threads within a particular SM can cooperate with each other (via data sharing) is through the shared memory (see Figure 19.8). This is typically accomplished by the programmer assigning only certain threads of an SM to write to specific locations of its shared memory, thus preventing write hazards or wasted cycles (e.g., many threads reading the same data

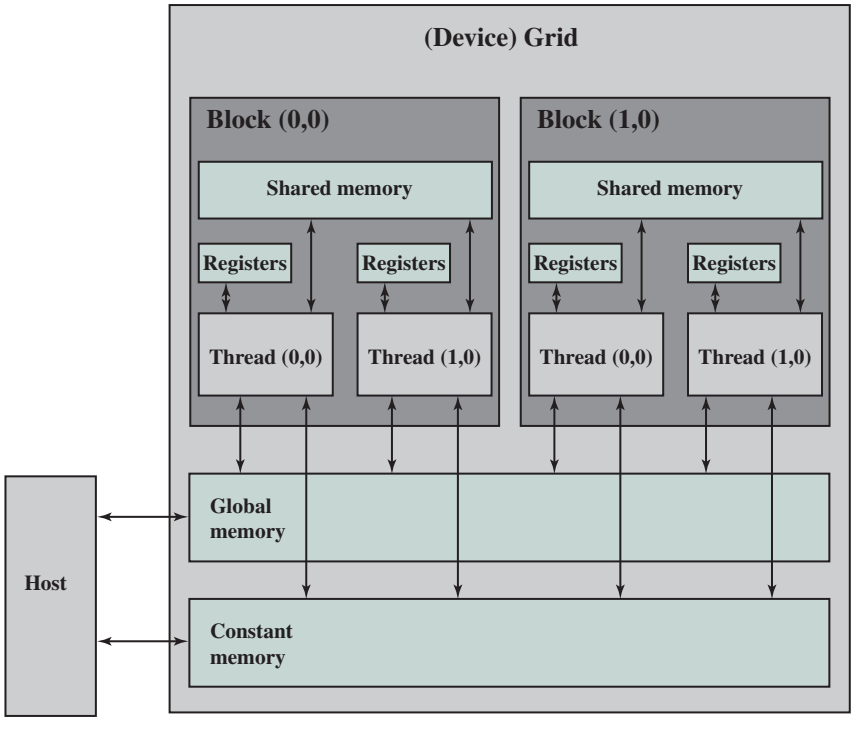


Figure 19.8 CUDA Representation of a GPU's Basic Architecture

from global memory and writing it to the same shared memory address). Before all of the threads of a particular SM are allowed to read from the shared memory that has just been written to, synchronization of all the threads of that SM needs to take place to prevent a read-after-write (RAW) data hazard.[1]

19.4 INTEL'S GEN8 GPU

As another example of a GPGPU architecture, this section provides an overview of the Gen8 processor graphics architecture [INTE14, PEDD14].

The fundamental building block of the Gen8 architecture is the execution unit (EU) shown in Figure 19.9. The EU is a simultaneous multithreading (SMT) architecture with seven threads. Recall from Chapters 17 and 18 that in an SMT architecture, register banks are expanded so that multiple threads can share the use of pipeline resources. The EU has seven threads and is implemented as a superscalar pipeline architecture. Each thread includes 128 general-purpose registers. Within each EU, the primary computation units are two SIMD floating-point units that support both floatingpoint and integer computation. Each SIMD FPU can complete simultaneous add and multiply floating-point instructions every cycle. There is also a branch unit dedicated to branch instructions and a send unit for memory operations.

Each register stores 32 bytes, accessible as an SIMD 8-element vector of 32-bit data elements. Thus each Gen8 thread has 4 kB of general-purpose register file (GRF), for a total of 28 kB of GRF per EU. Flexible addressing modes permit registers to be addressed together to build effectively wider registers, or even to represent strided rectangular block data structures.[2] Per thread architectural state is maintained in a separate dedicated architecture register file (ARF).

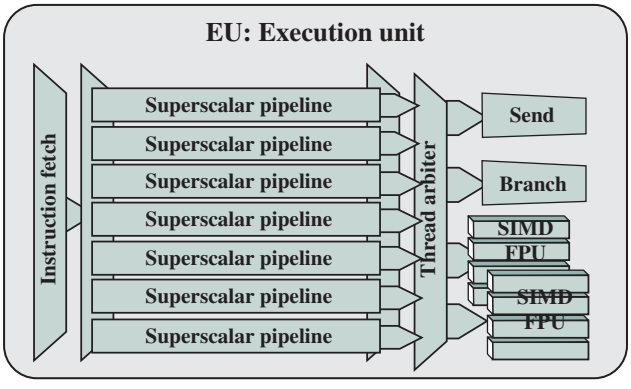


Figure 19.9 Intel Gen8 Execution Unit

[1]See Chapter 16 for a discussion of RAW hazards.

[2]The term *strided* refers to a sequence of memory reads and writes to addresses, each of which is separated from the last by a constant interval called the *stride length*. Strided references are often generated by loops through an array, and (if the data is large enough that access-time is significant) it can be worthwhile to tune for better locality by inverting double loops or by partially unrolling the outer loop of a loop nest.

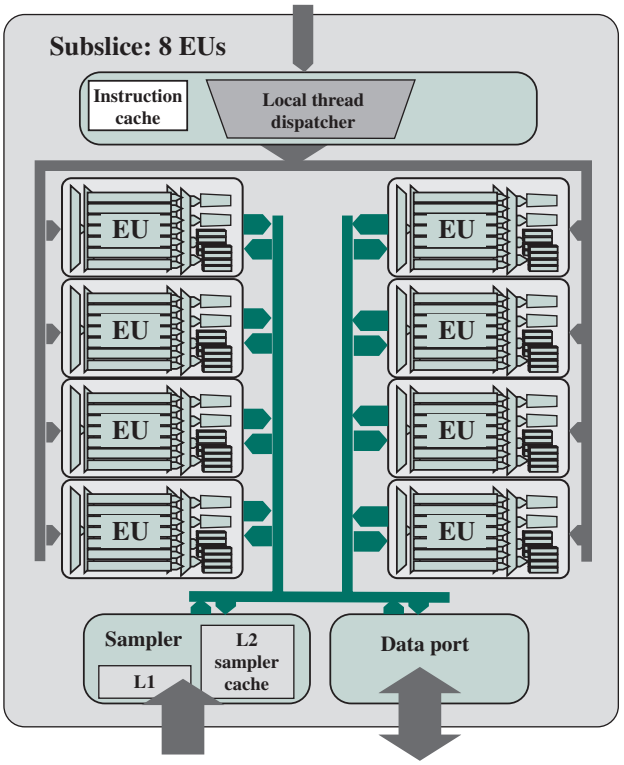


Figure 19.10 Intel Gen8 Subslice

The EU can issue up to four different instructions simultaneously from different threads. The thread arbiter dispatches each instruction to one of the four functional units for execution.

EUs are organized into a subslice (Figure 19.10), which may contain up to eight EUs. Each subslice contains its own local thread dispatcher unit and its own supporting instruction caches. Thus, a single subslice has dedicated hardware resources and register files for a total of 56 simultaneous threads.

A subslice also includes a unit called the sampler, with its own local L1 and L2 cache. The sampler is used for sampling texture and image surfaces. The sampler includes logic to support dynamic decompression of block compression texture formats. The sampler also includes fixed-function logic that enables address conversion of image (u,v) coordinates and address clamping modes such as mirror, wrap, border, and clamp. The sampler supports a variety of sampling filtering modes such as point, bilinear, trilinear, and anisotropic. The data port provides efficient read/write operations that attempt to take advantage of cache line size to consolidate read operations from different threads.

To create product variants, subslices may be clustered into groups called slices (Figure 19.11). Currently, up to three subslices may be organized into a single slice for a total of 24 EUs. In addition to the subslices, the slice includes logic for thread dispatch routing, other function logic to optimize graphic data processing, a shared

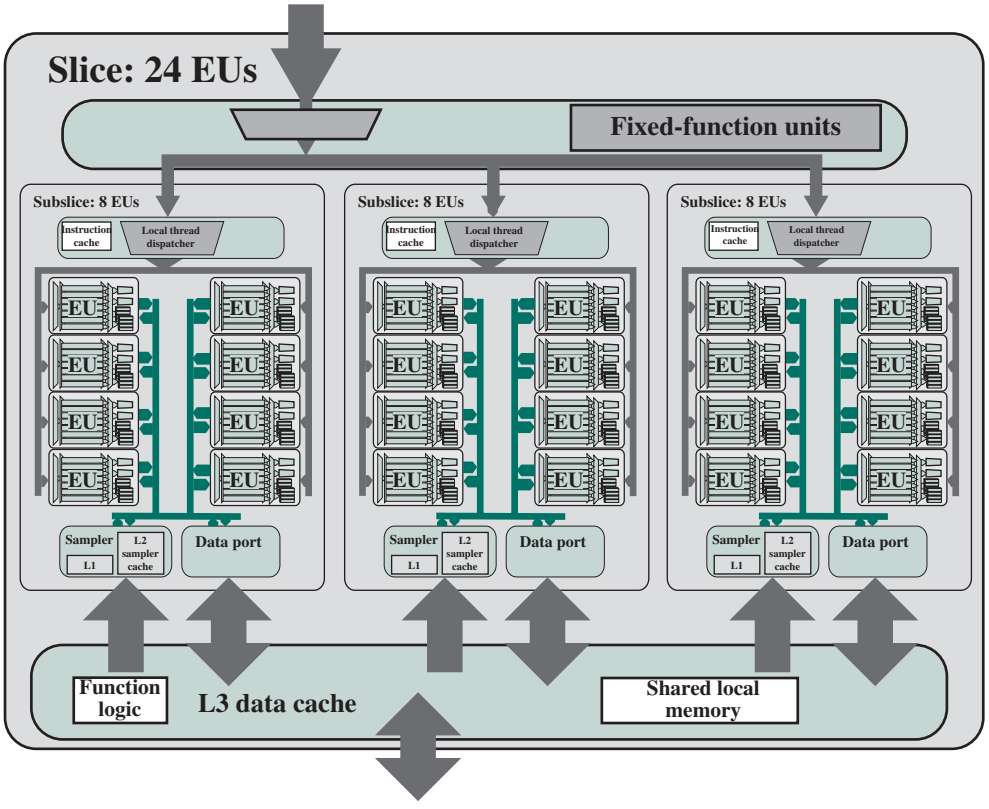


Figure 19.11 Intel Gen8 Slice

L3 cache, and a smaller shared local memory structure. The latter is visible (addressable memory) to the EUs and is useful for sharing temporary variables.

To enhance performance a technique known as **cache banking** is used for the shared L3 data cache. To achieve high bandwidth, the cache is divided into equal-size memory modules, called banks, which can be accessed simultaneously. Any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module. However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing throughput by a factor equal to the number of separate memory requests. If the number of separate memory requests is n, the initial memory request is said to cause n-way bank conflicts. To get maximum performance, it is therefore important to understand how memory addresses map to memory banks in order to schedule the memory requests so as to minimize bank conflicts.

Finally, an SoC product architect can create product families or a specific product within a family by placing a single slice or multiple slices on an SoC chip. These slices are combined with additional front-end logic to manage command

submission, as well as fixed-function logic to support 3D rendering and media pipelines. Additionally, the entire Gen8 compute architecture interfaces to the rest of the SoC components via a dedicated unit called the graphics technology interface (GTI).

An example of such an SoC is the Intel Core M Processor with Intel HD Graphics 5300 Gen8 (Figure 19.12). In addition to the GPU portion, the chip contains multiple CPU cores, an LLC cache and a system agent. The system agent includes controllers for DRAM memory, display, and PCIe devices. The Processor Graphics Gen8, CPUs, LLC cache, and system agent are interconnected with a ring structure, such as we saw for the Xeon processor (Figure 7.16).

19.5 WHEN TO USE A GPU AS A COPROCESSOR

We end this chapter with a brief discussion on determining candidate GPGPU applications from a software design perspective, as well as some related software tools to assist with this process.

What differentiates a program that would benefit from running a portion of its code on a GPU (thus, a heterogeneous computing platform) versus a program that wouldn't? As has been illustrated and discussed in this chapter, the GPU is made up of hundreds to thousands of processor cores and has an SIMD architecture. Therefore, programs that have a highly parallelizable portion(s) of code, which can be replicated into thousands of lightweight threads to work on large data sets concurrently, are the best candidates for accelerating their run time on a GPGPU system. Here, a lightweight thread is defined as an instance of a relatively small, massively parallelizable snippet of code, which has no or very little branching. Typically, the original serial code is in the form of a large iteration for-loop, or several embedded for-loops, which perform calculations on equations that have no data dependency between iterations (e.g., matrix arithmetic). Additionally, when initially profiling the entire program with tools similar to the GNU command line based gprof or NVIDIA's nvprof visual based profiler (either profiler preferably run against typical representative data), the section(s) to be parallelized must make up a fair percentage of the program's total run time. This requirement will both maximize the speedup that can be obtained (Amdahl's law) and minimize the impact that data transfer time between the CPU and the GPU will have on the overall speedup.

Once a candidate massively parallelizable code segment has been identified, it then needs to be converted from serial code to parallel code or a CUDA kernel. If a parallelizing compiler were available that could automatically do this conversion without input from the user and also give a near-optimal, correct solution, then that would save a great deal of time, money, and effort. Unfortunately, such a tool does not yet exist. This leaves two options: (1) convert the code through complex planning and programming in CUDA, OpenCL, or similar; or (2) use a compiler directive language, such as OpenACC, hiCUDA, or similar. Although using a compiler directive language to place parallelizing "hints" in the code for the compiler can save a great deal of programming time, it is still an iterative process and the optimum run time obtained is not guaranteed. However, this method has seen a

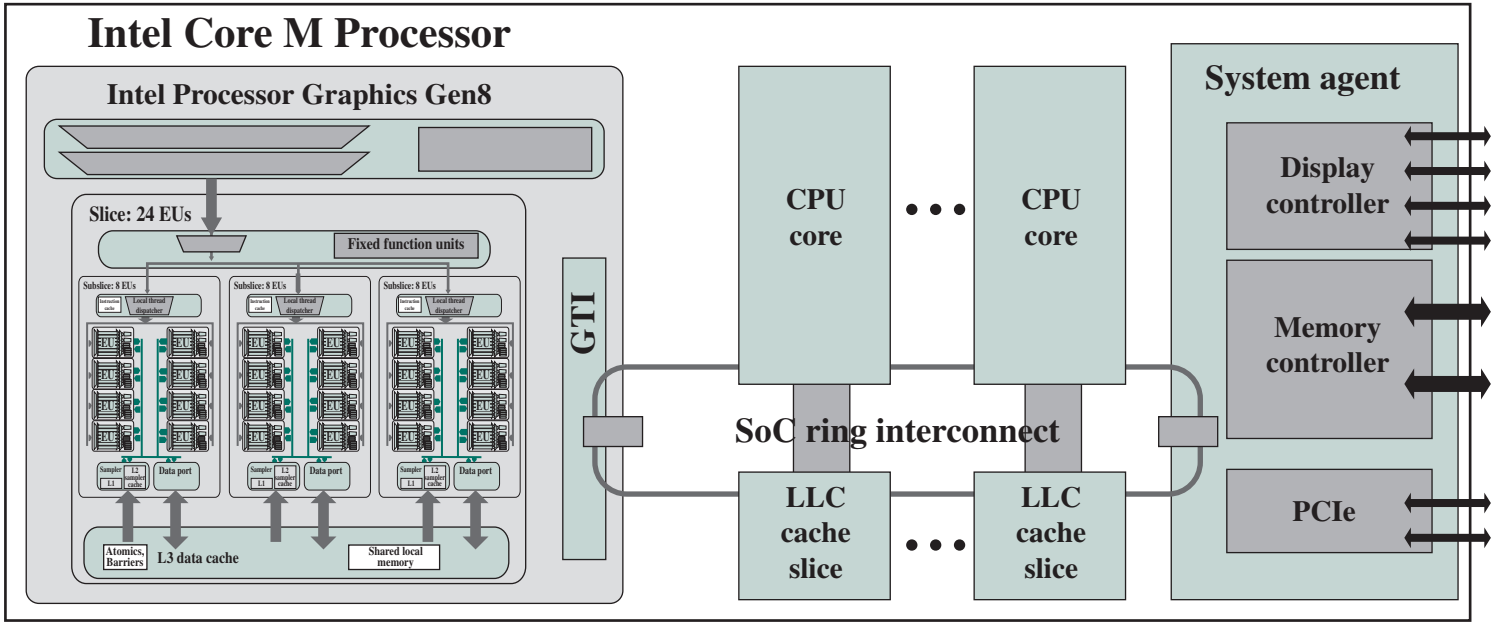


Figure 19.12 Intel Core M Processor SoC

growing interest over the past several years, and the newer versions of the CUDA compiler support the OpenACC language. Yet, a well-planned/engineered and coded CUDA program will almost always give the best runtimes to date.

19.6 KEY TERMS AND REVIEW QUESTIONS

Key Terms

block	general-purpose computing using a GPU (GPGPU)	kernel
cache banking	GPU processor core	streaming multiprocessors (SMs)
central processing unit (CPU)	graphic processing unit (GPU)	thread
Compute Unified Device Architecture (CUDA)	grid	thread block
CUDA core		warp

Review Questions

19.1 Define CUDA.
19.2 List the basic differences between CPU and GPU architectures.
19.3 What are the differences between kernel, thread, and block?
19.4 Define warp.
19.5 What are special function units?

PART SIX THE CONTROL UNIT

CHAPTER 20

CONTROL UNIT OPERATION

20.1 Micro-Operations
- The Fetch Cycle
- The Indirect Cycle
- The Interrupt Cycle
- The Execute Cycle
- The Instruction Cycle

20.2 Control of the Processor
- Functional Requirements
- Control Signals
- A Control Signals Example
- Internal Processor Organization
- The Intel 8085

20.3 Hardwired Implementation
- Control Unit Inputs
- Control Unit Logic

20.4 Key Terms, Review Questions, and Problems

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Explain the concept of micro-operations and define the principal instruction cycle phases in terms of micro-operations.
- Discuss how micro-operations are organized to control a processor.
- Understand hardwired control unit organization.

In Chapter 12, we pointed out that a machine instruction set goes a long way toward defining the processor. If we know the machine instruction set, including an understanding of the effect of each opcode and an understanding of the addressing modes, and if we know the set of user-visible registers, then we know the functions that the processor must perform. This is not the complete picture. We must know the external interfaces, usually through a bus, and how interrupts are handled. With this line of reasoning, the following list of those things needed to specify the function of a processor emerges:

1. Operations (opcodes)
2. Addressing modes
3. Registers
4. I/O module interface
5. Memory module interface
6. Interrupts

This list, though general, is rather complete. Items 1 through 3 are defined by the instruction set. Items 4 and 5 are typically defined by specifying the system bus. Item 6 is defined partially by the system bus and partially by the type of support the processor offers to the operating system.

This list of six items might be termed the functional requirements for a processor. They determine what a processor must do. This is what occupied us in Parts Two and Four. Now, we turn to the question of how these functions are performed or, more specifically, how the various elements of the processor are controlled to provide these functions. Thus, we turn to a discussion of the control unit, which controls the operation of the processor.

20.1 MICRO-OPERATIONS

We have seen that the operation of a computer, in executing a program, consists of a sequence of instruction cycles, with one machine instruction per cycle. Of course, we must remember that this sequence of instruction cycles is not necessarily the same as the *written sequence* of instructions that make up the program, because of the existence of branching instructions. What we are referring to here is the execution *time sequence* of instructions.

We have further seen that each instruction cycle is made up of a number of smaller units. One subdivision that we found convenient is fetch, indirect, execute, and interrupt, with only fetch and execute cycles always occurring.

To design a control unit, however, we need to break down the description further. In our discussion of pipelining in Chapter 14, we began to see that a further decomposition is possible. In fact, we will see that each of the smaller cycles involves a series of steps, each of which involves the processor registers. We will refer to these steps as **micro-operations**. The prefix *micro* refers to the fact that each step is very simple and accomplishes very little. Figure 20.1 depicts the relationship among the various concepts we have been discussing. To summarize, the execution of a program consists of the sequential execution of instructions. Each instruction is executed during an instruction cycle made up of shorter subcycles (e.g., fetch, indirect, execute, interrupt). The execution of each subcycle involves one or more shorter operations, that is, micro-operations.

Micro-operations are the functional, or atomic, operations of a processor. In this section, we will examine micro-operations to gain an understanding of how the events of any instruction cycle can be described as a sequence of such micro-operations. A simple example will be used. In the remainder of this chapter, we then show how the concept of micro-operations serves as a guide to the design of the control unit.

The Fetch Cycle

We begin by looking at the fetch cycle, which occurs at the beginning of each instruction cycle and causes an instruction to be fetched from memory. For purposes of discussion, we assume the organization depicted in Figure 14.6 (*Data Flow*, *Fetch Cycle*). Four registers are involved:

- **Memory address register (MAR):** Is connected to the address lines of the system bus. It specifies the address in memory for a read or write operation.
- **Memory buffer register (MBR):** Is connected to the data lines of the system bus. It contains the value to be stored in memory or the last value read from memory.

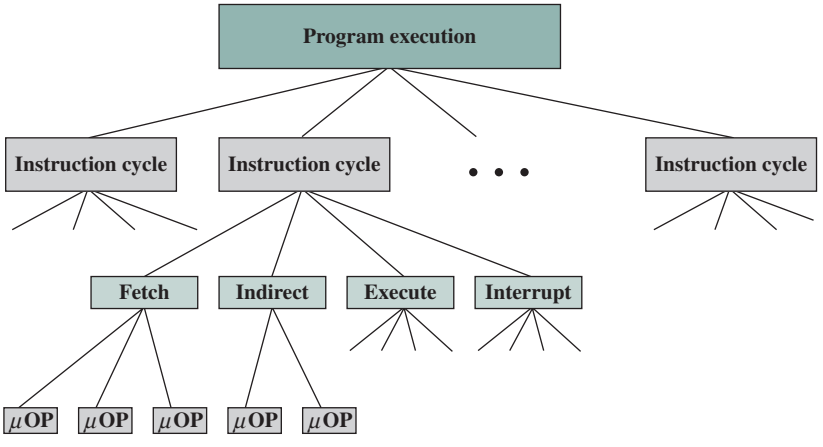


Figure 20.1 Constituent Elements of a Program Execution

- **Program counter (PC):** Holds the address of the next instruction to be fetched.
- **Instruction register (IR):** Holds the last instruction fetched.

Let us look at the sequence of events for the fetch cycle from the point of view of its effect on the processor registers. An example appears in Figure 20.2. At the beginning of the fetch cycle, the address of the next instruction to be executed is in the program counter (PC); in this case, the address is 1100100. The first step is to move that address to the memory address register (MAR) because this is the only register connected to the address lines of the system bus. The second step is to bring in the instruction. The desired address (in the MAR) is placed on the address bus, the control unit issues a READ command on the control bus, and the result appears on the data bus and is copied into the memory buffer register (MBR). We also need to increment the PC by the instruction length to get ready for the next instruction. Because these two actions (read word from memory, increment PC) do not interfere with each other, we can do them simultaneously to save time. The third step is to move the contents of the MBR to the instruction register (IR). This frees up the MBR for use during a possible indirect cycle.

Thus, the simple fetch cycle actually consists of three steps and four micro-operations. Each micro-operation involves the movement of data into or out of a register. So long as these movements do not interfere with one another, several of them can take place during one step, saving time. Symbolically, we can write this sequence of events as follows:

t_1: MAR ← (PC)
t_2: MBR ← Memory
PC ← (PC) + I
t_3: IR ← (MBR)

where I is the instruction length. We need to make several comments about this sequence. We assume that a clock is available for timing purposes and that it emits regularly spaced clock pulses. Each clock pulse defines a time unit. Thus, all time units are

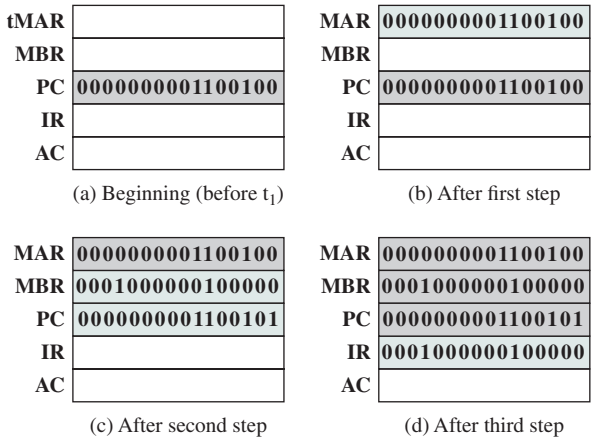


Figure 20.2 Sequence of Events, Fetch Cycle

of equal duration. Each micro-operation can be performed within the time of a single time unit. The notation (t_1, t_2, t_3) represents successive time units. In words, we have

- **First time unit:** Move contents of PC to MAR.
- **Second time unit:** Move contents of memory location specified by MAR to MBR. Increment by *I* the contents of the PC.
- **Third time unit:** Move contents of MBR to IR.

Note that the second and third micro-operations both take place during the second time unit. The third micro-operation could have been grouped with the fourth without affecting the fetch operation:

```
t1: MAR  ← (PC)
t2: MBR  ← Memory
t3: PC   ← (PC) + I
    IR   ← (MBR)
```

The groupings of micro-operations must follow two simple rules:

1. The proper sequence of events must be followed. Thus (MAR ← (PC)) must precede (MBR ← Memory) because the memory read operation makes use of the address in the MAR.
2. Conflicts must be avoided. One should not attempt to read to and write from the same register in one time unit, because the results would be unpredictable. For example, the micro-operations (MBR ← Memory) and (IR ← MBR) should not occur during the same time unit.

A final point worth noting is that one of the micro-operations involves an addition. To avoid duplication of circuitry, this addition could be performed by the ALU. The use of the ALU may involve additional micro-operations, depending on the functionality of the ALU and the organization of the processor. We defer a discussion of this point until later in this chapter.

It is useful to compare events described in this and the following subsections to Figure 3.5 (*Example of Program Execution*). Whereas micro-operations are ignored in that figure, this discussion shows the micro-operations needed to perform the subcycles of the instruction cycle.

The Indirect Cycle

Once an instruction is fetched, the next step is to fetch source operands. Continuing our simple example, let us assume a one-address instruction format, with direct and indirect addressing allowed. If the instruction specifies an indirect address, then an indirect cycle must precede the execute cycle. The data flow differs somewhat from that indicated in Figure 14.7 (*Data Flow*, *Indirect Cycle*) and includes the following micro-operations:

```
t1: MAR  ← (IR(Address))
t2: MBR  ← Memory
t3: IR(Address)  ← (MBR(Address))
```

The address field of the instruction is transferred to the MAR. This is then used to fetch the address of the operand. Finally, the address field of the IR is updated from the MBR, so that it now contains a direct rather than an indirect address.

The IR is now in the same state as if indirect addressing had not been used, and it is ready for the execute cycle. We skip that cycle for a moment, to consider the interrupt cycle.

The Interrupt Cycle

At the completion of the execute cycle, a test is made to determine whether any enabled interrupts have occurred. If so, the interrupt cycle occurs. The nature of this cycle varies greatly from one machine to another. We present a very simple sequence of events, as illustrated in Figure 14.8 (*Data Flow, Indirect Cycle*). We have

```
t1: MBR ← (PC)
t2: MAR ← Save_Address
    PC ← Routine_Address
t3: Memory ← (MBR)
```

In the first step, the contents of the PC are transferred to the MBR, so that they can be saved for return from the interrupt. Then the MAR is loaded with the address at which the contents of the PC are to be saved, and the PC is loaded with the address of the start of the interrupt-processing routine. These two actions may each be a single micro-operation. However, because most processors provide multiple types and/or levels of interrupts, it may take one or more additional micro-operations to obtain the Save_Address and the Routine_Address before they can be transferred to the MAR and PC, respectively. In any case, once this is done, the final step is to store the MBR, which contains the old value of the PC, into memory. The processor is now ready to begin the next instruction cycle.

The Execute Cycle

The fetch, indirect, and interrupt cycles are simple and predictable. Each involves a small, fixed sequence of micro-operations and, in each case, the same micro-operations are repeated each time around.

This is not true of the execute cycle. Because of the variety of opcodes, there are a number of different sequences of micro-operations that can occur. The control unit examines the opcode and generates a sequence of micro-operations based on the value of the opcode. This is referred to as instruction decoding.

Let us consider several hypothetical examples.

First, consider an add instruction:

```
ADD R1, X
```

which adds the contents of the location X to register R1. The following sequence of micro-operations might occur:

```
t1: MAR ← (IR(address))
t2: MBR ← Memory
t3: R1  ← (R1) + (MBR)
```

We begin with the IR containing the ADD instruction. In the first step, the address portion of the IR is loaded into the MAR. Then the referenced memory location is read. Finally, the contents of R1 and MBR are added by the ALU. Again, this is a simplified example. Additional micro-operations may be required to extract

the register reference from the IR and perhaps to stage the ALU inputs or outputs in some intermediate registers.

Let us look at two more complex examples. A common instruction is increment and skip if zero:

```
ISZ X
```

The content of location X is incremented by 1. If the result is 0, the next instruction is skipped. A possible sequence of micro-operations is

```
t1: MAR ← (IR(address))
t2: MBR ← Memory
t3: MBR ← (MBR) + 1
t4: Memory ← (MBR)
    If ((MBR) = 0) then (PC ← (PC) + I)
```

The new feature introduced here is the conditional action. The PC is incremented if (MBR) = 0. This test and action can be implemented as one micro-operation. Note also that this micro-operation can be performed during the same time unit during which the updated value in MBR is stored back to memory.

Finally, consider a subroutine call instruction. As an example, consider a branch-and-save-address instruction:

```
BSA X
```

The address of the instruction that follows the BSA instruction is saved in location X, and execution continues at location X + I. The saved address will later be used for return. This is a straightforward technique for supporting subroutine calls. The following micro-operations suffice:

```
t1: MAR ← (IR(address))
    MBR ← (PC)
t2: PC ← (IR(address))
    Memory ← (MBR)
t3: PC ← (PC) + I
```

The address in the PC at the start of the instruction is the address of the next instruction in sequence. This is saved at the address designated in the IR. The latter address is also incremented to provide the address of the instruction for the next instruction cycle.

The Instruction Cycle

We have seen that each phase of the instruction cycle can be decomposed into a sequence of elementary micro-operations. In our example, there is one sequence each for the fetch, indirect, and interrupt cycles, and, for the execute cycle, there is one sequence of micro-operations for each opcode.

To complete the picture, we need to tie sequences of micro-operations together, and this is done in Figure 20.3. We assume a new 2-bit register called the *instruction cycle code* (ICC). The ICC designates the state of the processor in terms of which portion of the cycle it is in:

00: Fetch
01: Indirect

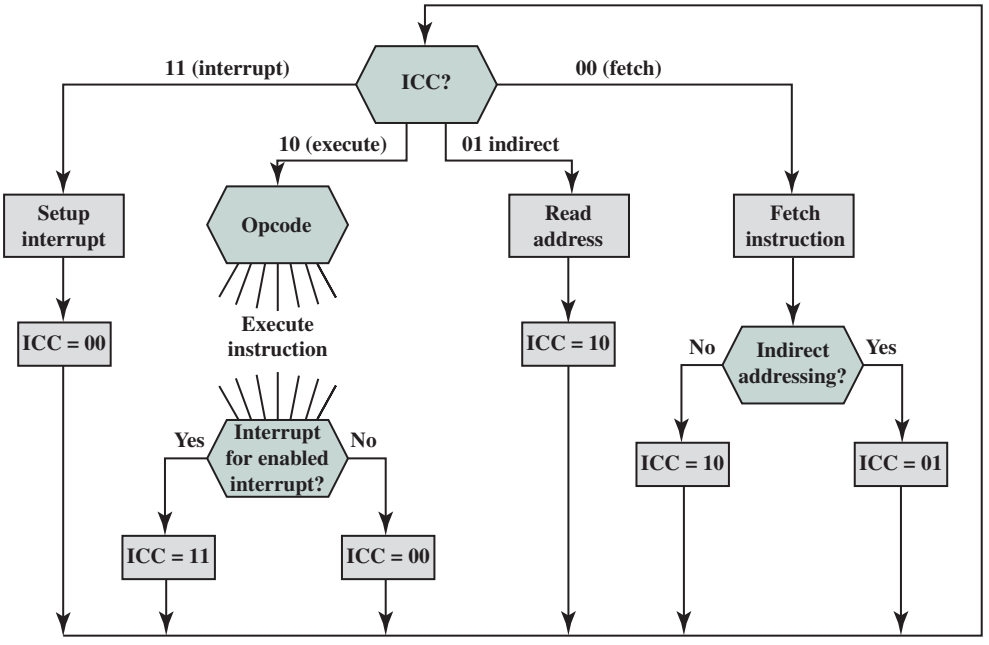


Figure 20.3 Flowchart for Instruction Cycle

10: Execute
11: Interrupt

At the end of each of the four cycles, the ICC is set appropriately. The indirect cycle is always followed by the execute cycle. The interrupt cycle is always followed by the fetch cycle (see Figure 14.4, *The Instruction Cycle*). For both the fetch and execute cycles, the next cycle depends on the state of the system.

Thus, the flowchart of Figure 20.3 defines the complete sequence of micro-operations, depending only on the instruction sequence and the interrupt pattern. Of course, this is a simplified example. The flowchart for an actual processor would be more complex. In any case, we have reached the point in our discussion in which the operation of the processor is defined as the performance of a sequence of micro-operations. We can now consider how the control unit causes this sequence to occur.

20.2 CONTROL OF THE PROCESSOR

Functional Requirements

As a result of our analysis in the preceding section, we have decomposed the behavior or functioning of the processor into elementary operations, called micro-operations. By reducing the operation of the processor to its most fundamental level, we are able to define exactly what it is that the control unit must cause to happen. Thus, we can define the *functional requirements* for the control unit: those functions that the control unit must perform. A definition of these functional requirements is the basis for the design and implementation of the control unit.

With the information at hand, the following three-step process leads to a characterization of the control unit:

1. Define the basic elements of the processor.
2. Describe the micro-operations that the processor performs.
3. Determine the functions that the control unit must perform to cause the micro-operations to be performed.

We have already performed steps 1 and 2. Let us summarize the results. First, the basic functional elements of the processor are the following:

- ALU
- Registers
- Internal data paths
- External data paths
- Control unit

Some thought should convince you that this is a complete list. The ALU is the functional essence of the computer. Registers are used to store data internal to the processor. Some registers contain status information needed to manage instruction sequencing (e.g., a program status word). Others contain data that go to or come from the ALU, memory, and I/O modules. Internal data paths are used to move data between registers and between register and ALU. External data paths link registers to memory and I/O modules, often by means of a system bus. The control unit causes operations to happen within the processor.

The execution of a program consists of operations involving these processor elements. As we have seen, these operations consist of a sequence of micro-operations. Upon review of Section 20.1, the reader should see that all micro-operations fall into one of the following categories:

- Transfer data from one register to another.
- Transfer data from a register to an external interface (e.g., system bus).
- Transfer data from an external interface to a register.
- Perform an arithmetic or logic operation, using registers for input and output.

All of the micro-operations needed to perform one instruction cycle, including all of the micro-operations to execute every instruction in the instruction set, fall into one of these categories.

We can now be somewhat more explicit about the way in which the control unit functions. The control unit performs two basic tasks:

- **Sequencing:** The control unit causes the processor to step through a series of micro-operations in the proper sequence, based on the program being executed.
- **Execution:** The control unit causes each micro-operation to be performed.

The preceding is a functional description of what the control unit does. The key to how the control unit operates is the use of control signals.

Control Signals

We have defined the elements that make up the processor (ALU, registers, data paths) and the micro-operations that are performed. For the control unit to perform its function, it must have inputs that allow it to determine the state of the system and outputs that allow it to control the behavior of the system. These are the external specifications of the control unit. Internally, the control unit must have the logic required to perform its sequencing and execution functions. We defer a discussion of the internal operation of the control unit to Section 20.3 and Chapter 21. The remainder of this section is concerned with the interaction between the control unit and the other elements of the processor.

Figure 20.4 is a general model of the control unit, showing all of its inputs and outputs. The inputs are:

- **Clock:** This is how the control unit "keeps time." The control unit causes one micro-operation (or a set of simultaneous micro-operations) to be performed for each clock pulse. This is sometimes referred to as the processor cycle time, or the clock cycle time.
- **Instruction register:** The opcode and addressing mode of the current instruction are used to determine which micro-operations to perform during the execute cycle.
- **Flags:** These are needed by the control unit to determine the status of the processor and the outcome of previous ALU operations. For example, for the increment-and-skip-if-zero (ISZ) instruction, the control unit will increment the PC if the zero flag is set.
- **Control signals from control bus:** The control bus portion of the system bus provides signals to the control unit.

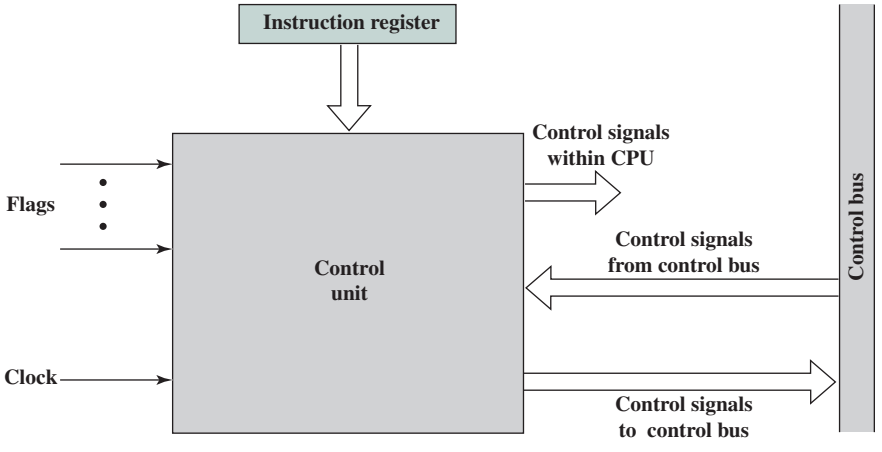


Figure 20.4 Block Diagram of the Control Unit

The outputs are as follows:

- **Control signals within the processor:** These are two types: those that cause data to be moved from one register to another, and those that activate specific ALU functions.
- **Control signals to control bus:** These are also of two types: control signals to memory, and control signals to the I/O modules.

Three types of control signals are used: those that activate an ALU function; those that activate a data path; and those that are signals on the external system bus or other external interface. All of these signals are ultimately applied directly as binary inputs to individual logic gates.

Let us consider again the fetch cycle to see how the control unit maintains control. The control unit keeps track of where it is in the instruction cycle. At a given point, it knows that the fetch cycle is to be performed next. The first step is to transfer the contents of the PC to the MAR. The control unit does this by activating the control signal that opens the gates between the bits of the PC and the bits of the MAR. The next step is to read a word from memory into the MBR and increment the PC. The control unit does this by sending the following control signals simultaneously:

- A control signal that opens gates, allowing the contents of the MAR onto the address bus;
- A memory read control signal on the control bus;
- A control signal that opens the gates, allowing the contents of the data bus to be stored in the MBR;
- Control signals to logic that add 1 to the contents of the PC and store the result back to the PC.

Following this, the control unit sends a control signal that opens gates between the MBR and the IR.

This completes the fetch cycle except for one thing: The control unit must decide whether to perform an indirect cycle or an execute cycle next. To decide this, it examines the IR to see if an indirect memory reference is made.

The indirect and interrupt cycles work similarly. For the execute cycle, the control unit begins by examining the opcode and, on the basis of that, decides which sequence of micro-operations to perform for the execute cycle.

A Control Signals Example

To illustrate the functioning of the control unit, let us examine a simple example. Figure 20.5 illustrates the example. This is a simple processor with a single accumulator (AC). The data paths between elements are indicated. The control paths for signals emanating from the control unit are not shown, but the terminations of control signals are labeled C_i and indicated by a circle. The control unit receives inputs from the clock, the IR, and flags. With each clock cycle, the control unit

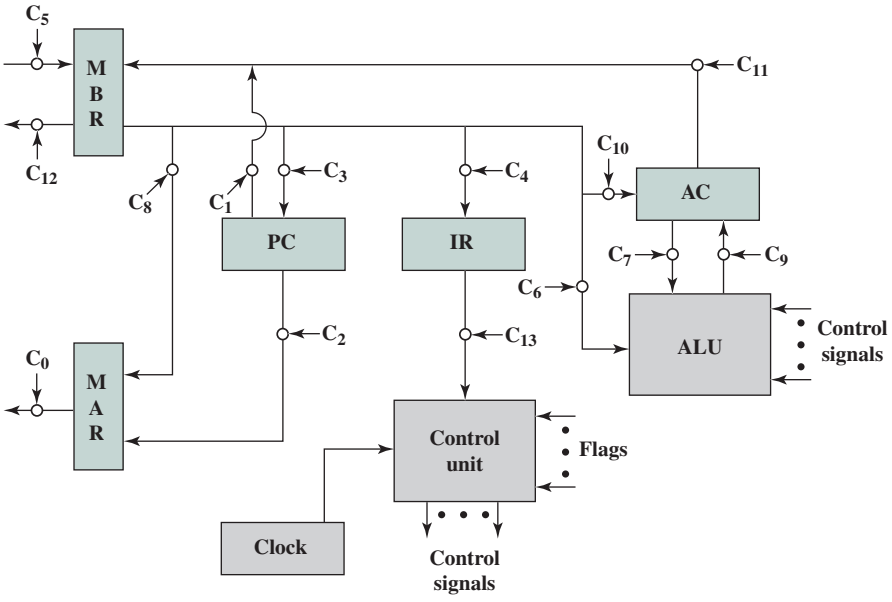


Figure 20.5 Data Paths and Control Signals

reads all of its inputs and emits a set of control signals. Control signals go to three separate destinations:

- **Data paths:** The control unit controls the internal flow of data. For example, on instruction fetch, the contents of the memory buffer register are transferred to the IR. For each path to be controlled, there is a switch (indicated by a circle in the figure). A control signal from the control unit temporarily opens the gate to let data pass.
- **ALU:** The control unit controls the operation of the ALU by a set of control signals. These signals activate various logic circuits and gates within the ALU.
- **System bus:** The control unit sends control signals out onto the control lines of the system bus (e.g., memory READ).

The control unit must maintain knowledge of where it is in the instruction cycle. Using this knowledge, and by reading all of its inputs, the control unit emits a sequence of control signals that causes micro-operations to occur. It uses the clock pulses to time the sequence of events, allowing time between events for signal levels to stabilize. Table 20.1 indicates the control signals that are needed for some of the micro-operation sequences described earlier. For simplicity, the data and control paths for incrementing the PC and for loading the fixed addresses into the PC and MAR are not shown.

It is worth pondering the minimal nature of the control unit. The control unit is the engine that runs the entire computer. It does this based only on knowing the instructions to be executed and the nature of the results of arithmetic and logical operations (e.g., positive, overflow, etc.). It never gets to see the data being processed or the actual results produced. And it controls everything with a few control signals to points within the processor and a few control signals to the system bus.

Table 20.1 Micro-operations and Control Signals

	Micro-operations	Active Control Signals
Fetch:	t_1: MAR ← (PC)	C_2
	t_2: MBR ← Memory PC ← (PC) + 1	C_5, C_R
	t_3: IR ← (MBR)	C_4
Indirect:	t_1: MAR ← (IR(Address))	C_8
	t_2: MBR ← Memory	C_5, C_R
	t_3: IR(Address) ← (MBR(Address))	C_4
Interrupt:	t_1: MBR ← (PC)	C_1
	t_2: MAR ← Save-address PC ← Routine-address	
	t_3: Memory ← (MBR)	C_{12}, C_W

C_R = Read control signal to system bus.

C_W = Write control signal to system bus.

Internal Processor Organization

Figure 20.5 indicates the use of a variety of data paths. The complexity of this type of organization should be clear. More typically, some sort of internal bus arrangement, as was suggested in Figure 14.2 (*Internal Structure of the CPU*), will be used.

Using an internal processor bus, Figure 20.5 can be rearranged as shown in Figure 20.6. A single internal bus connects the ALU and all processor registers. Gates and control signals are provided for movement of data onto and off the bus from each register. Additional control signals control data transfer to and from the system (external) bus and the operation of the ALU.

Two new registers, labeled Y and Z, have been added to the organization. These are needed for the proper operation of the ALU. When an operation involving two operands is performed, one can be obtained from the internal bus, but the other must be obtained from another source. The AC could be used for this purpose, but this limits the flexibility of the system and would not work with a processor with multiple general-purpose registers. Register Y provides temporary storage for the other input. The ALU is a combinatorial circuit (see Chapter 11) with no internal storage. Thus, when control signals activate an ALU function, the input to the ALU is transformed to the output. Therefore, the output of the ALU cannot be directly connected to the bus, because this output would feed back to the input. Register Z provides temporary output storage. With this arrangement, an operation to add a value from memory to the AC would have the following steps:

```
t1: MAR ← (IR(address))
t2: MBR ← Memory
t3: Y ← (MBR)
t4: Z ← (AC) + (Y)
t5: AC ← (Z)
```

Other organizations are possible, but, in general, some sort of internal bus or set of internal buses is used. The use of common data paths simplifies the

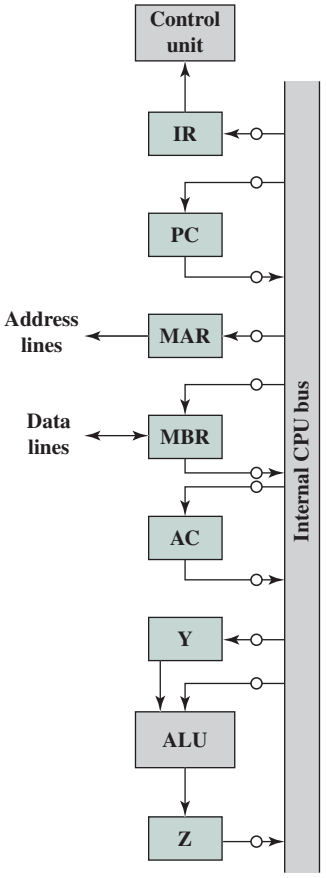


Figure 20.6 CPU with Internal Bus

interconnection layout and the control of the processor. Another practical reason for the use of an internal bus is to save space.

The Intel 8085

To illustrate some of the concepts introduced thus far in this chapter, let us consider the Intel 8085. Its organization is shown in Figure 20.7. Several key components that may not be self-explanatory are:

- **Incrementer/decrementer address latch:** Logic that can add 1 to or subtract 1 from the contents of the stack pointer or program counter. This saves time by avoiding the use of the ALU for this purpose.
- **Interrupt control:** This module handles multiple levels of interrupt signals.
- **Serial I/O control:** This module interfaces to devices that communicate 1 bit at a time.

Table 20.2 describes the external signals into and out of the 8085. These are linked to the external system bus. These signals are the interface between the 8085 processor and the rest of the system (Figure 20.8).

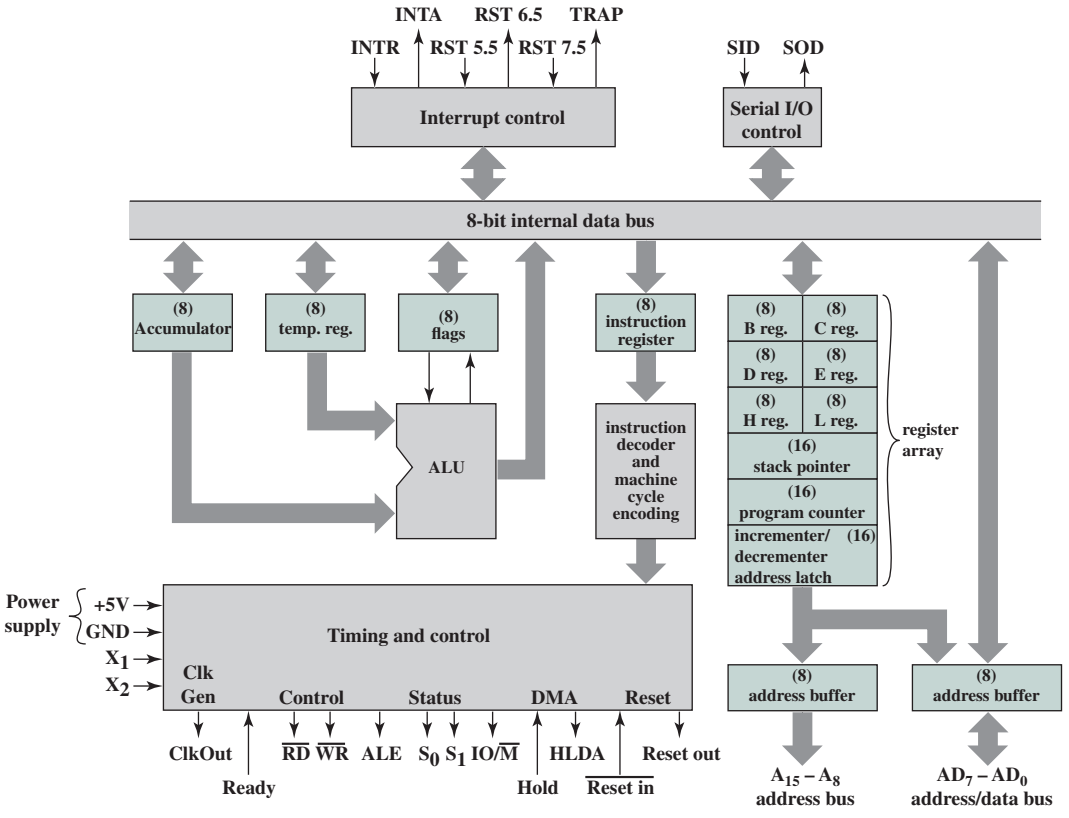


Figure 20.7 Intel 8085 CPU Block Diagram

The control unit is identified as having two components labeled (1) instruction decoder and machine cycle encoding and (2) timing and control. A discussion of the first component is deferred until the next section. The essence of the control unit is the timing and control module. This module includes a clock and accepts as inputs the current instruction and some external control signals. Its output consists of control signals to the other components of the processor plus control signals to the external system bus.

The timing of processor operations is synchronized by the clock and controlled by the control unit with control signals. Each instruction cycle is divided into from one to five *machine cycles*; each machine cycle is in turn divided into from three to five *states.* Each state lasts one clock cycle. During a state, the processor performs one or a set of simultaneous micro-operations as determined by the control signals.

The number of machine cycles is fixed for a given instruction but varies from one instruction to another. Machine cycles are defined to be equivalent to bus accesses. Thus, the number of machine cycles for an instruction depends on the number of times the processor must communicate with external devices. For example, if an instruction consists of two 8-bit portions, then two machine cycles are required to fetch the instruction. If that instruction involves a 1-byte memory or I/O operation, then a third machine cycle is required for execution.

Table 20.2 Intel 8085 External Signals

Address and Data Signals

High Address (A15–A8)
The high-order 8 bits of a 16-bit address.

Address/Data (AD7–AD0)
The lower-order 8 bits of a 16-bit address or 8 bits of data. This multiplexing saves on pins.

Serial Input Data (SID)
A single-bit input to accommodate devices that transmit serially (one bit at a time).

Serial Output Data (SOD)
A single-bit output to accommodate devices that receive serially.

Timing and Control Signals

CLK (OUT)
The system clock. The CLK signal goes to peripheral chips and synchronizes their timing.

X1, X2
These signals come from an external crystal or other device to drive the internal clock generator.

Address Latch Enabled (ALE)
Occurs during the first clock state of a machine cycle and causes peripheral chips to store the address lines. This allows the address module (e.g., memory, I/O) to recognize that it is being addressed.

Status (S0, S1)
Control signals used to indicate whether a read or write operation is taking place.

IO/M
Used to enable either I/O or memory modules for read and write operations.

Read Control (RD)
Indicates that the selected memory or I/O module is to be read and that the data bus is available for data transfer.

Write Control (WR)
Indicates that data on the data bus is to be written into the selected memory or I/O location.

Memory and I/O Initiated Symbols

Hold
Requests the CPU to relinquish control and use of the external system bus. The CPU will complete execution of the instruction presently in the IR and then enter a hold state, during which no signals are inserted by the CPU to the control, address, or data buses. During the hold state, the bus may be used for DMA operations.

Hold Acknowledge (HOLDA)
This control unit output signal acknowledges the HOLD signal and indicates that the bus is now available.

READY
Used to synchronize the CPU with slower memory or I/O devices. When an addressed device asserts READY, the CPU may proceed with an input (DBIN) or output (WR) operation. Otherwise, the CPU enters a wait state until the device is ready.

Interrupt-Related Signals

TRAP
Restart Interrupts (RST 7.5, 6.5, 5.5)

Interrupt Request (INTR)
These five lines are used by an external device to interrupt the CPU. The CPU will not honor the request if it is in the hold state or if the interrupt is disabled. An interrupt is honored only at the completion of an instruction. The interrupts are in descending order of priority.

Interrupt Acknowledge
Acknowledges an interrupt.

CPU Initialization

RESET IN
Causes the contents of the PC to be set to zero. The CPU resumes execution at location zero.

RESET OUT
Acknowledges that the CPU has been reset. The signal can be used to reset the rest of the system.

Voltage and Ground

VCC
+5-volt power supply

VSS
Electrical ground

Figure 20.9 gives an example of 8085 timing, showing the value of external control signals. Of course, at the same time, the control unit generates internal control signals that control internal data transfers. The diagram shows the instruction cycle for an OUT instruction. Three machine cycles (M_1, M_2, M_3) are needed. During the first, the OUT instruction is fetched. The second machine cycle fetches the second half of the instruction, which contains the number of the I/O device selected for output. During the third cycle, the contents of the AC are written out to the selected device over the data bus.

The Address Latch Enabled (ALE) pulse signals the start of each machine cycle from the control unit. The ALE pulse alerts external circuits. During timing state T_1 of machine cycle M_1, the control unit sets the IO/M signal to indicate that this is a memory operation. Also, the control unit causes the contents of the PC

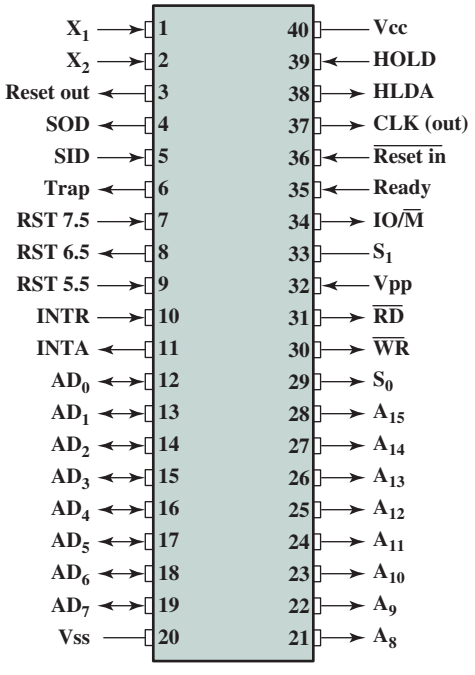


Figure 20.8 Intel 8085 Pin Configuration

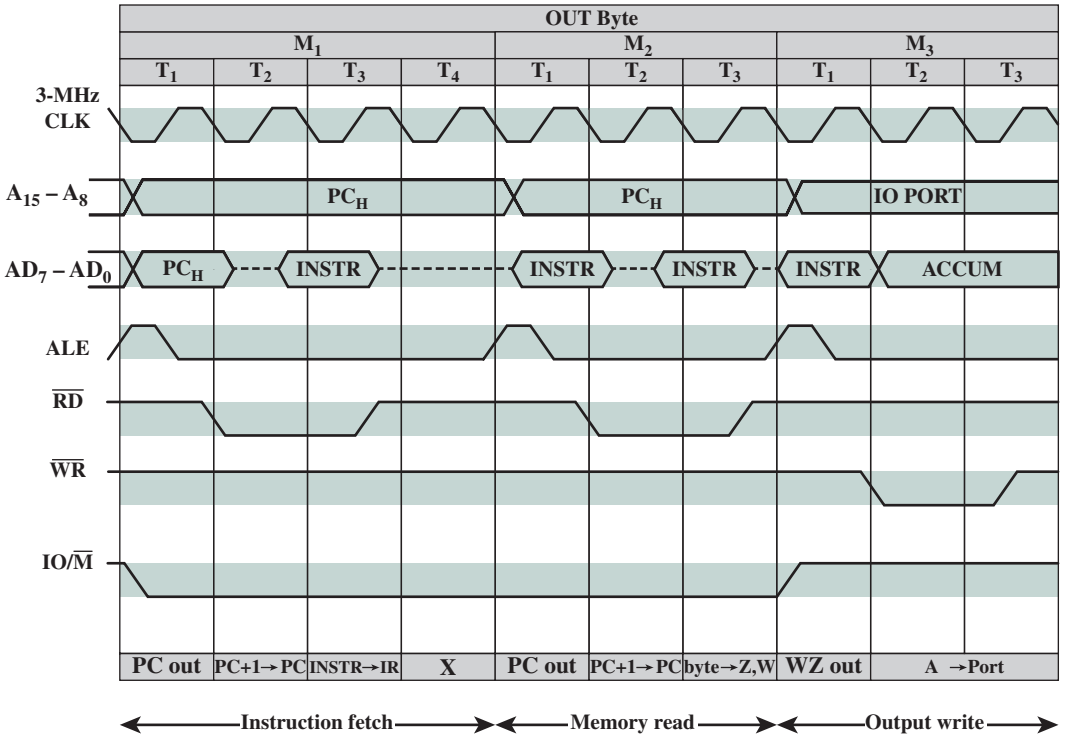


Figure 20.9 Timing Diagram for Intel 8085 OUT Instruction

to be placed on the address bus (A_{15} through A_8) and the address/data bus (AD_7 through AD_0). With the falling edge of the ALE pulse, the other modules on the bus store the address.

During timing state T_2, the addressed memory module places the contents of the addressed memory location on the address/data bus. The control unit sets the Read Control (RD) signal to indicate a read, but it waits until T_3 to copy the data from the bus. This gives the memory module time to put the data on the bus and for the signal levels to stabilize. The final state, T_4, is a *bus idle* state during which the processor decodes the instruction. The remaining machine cycles proceed in a similar fashion.

20.3 HARDWIRED IMPLEMENTATION

We have discussed the control unit in terms of its inputs, output, and functions. We now turn to the topic of control unit implementation. A wide variety of techniques have been used. Most of these fall into one of two categories:

- Hardwired implementation
- Microprogrammed implementation

In a **hardwired implementation**, the control unit is essentially a state machine circuit. Its input logic signals are transformed into a set of output logic signals, which

are the control signals. This approach is examined in this section. Microprogrammed implementation is the subject of Chapter 21.

Control Unit Inputs

Figure 20.4 depicts the control unit as we have so far discussed it. The key inputs are the IR, the clock, flags, and control bus signals. In the case of the flags and control bus signals, each individual bit typically has some meaning (e.g., overflow). The other two inputs, however, are not directly useful to the control unit.

First consider the IR. The control unit makes use of the opcode and will perform different actions (issue a different combination of control signals) for different instructions. To simplify the control unit logic, there should be a unique logic input for each opcode. This function can be performed by a *decoder*, which takes an encoded input and produces a single output. In general, a decoder will have n binary inputs and 2^n binary outputs. Each of the 2^n different input patterns will activate a single unique output. Table 20.3 is an example for $n = 4$. The decoder for a control unit will typically have to be more complex than that, to account for variable-length opcodes. An example of the digital logic used to implement a decoder is presented in Chapter 11.

The clock portion of the control unit issues a repetitive sequence of pulses. This is useful for measuring the duration of micro-operations. Essentially, the period of the clock pulses must be long enough to allow the propagation of signals along

Table 20.3 A Decoder with 4 Inputs and 16 Outputs

I1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
I2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
I3	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
I4	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
O1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
O2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
O3	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
O4	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
O5	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
O6	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
O7	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
O8	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
O9	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
O10	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
O11	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
O12	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
O13	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
O14	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
O15	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
O16	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

data paths and through processor circuitry. However, as we have seen, the control unit emits different control signals at different time units within a single instruction cycle. Thus, we would like a counter as input to the control unit, with a different control signal being used for T_1,T_2, and so forth. At the end of an instruction cycle, the control unit must feed back to the counter to reinitialize it at T_1.

With these two refinements, the control unit can be depicted as in Figure 20.10.

Control Unit Logic

To define the hardwired implementation of a control unit, all that remains is to discuss the internal logic of the control unit that produces output control signals as a function of its input signals.

Essentially, what must be done is, for each control signal, to derive a Boolean expression of that signal as a function of the inputs. This is best explained by example. Let us consider again our simple example illustrated in Figure 20.5. We saw in Table 20.1 the micro-operation sequences and control signals needed to control three of the four phases of the instruction cycle.

Let us consider a single control signal, C_5. This signal causes data to be read from the external data bus into the MBR. We can see that it is used twice in Table 20.1. Let us define two new control signals, P and Q, that have the following interpretation:

PQ = 00	Fetch Cycle
PQ = 01	Indirect Cycle
PQ = 10	Execute Cycle
PQ = 11	Interrupt Cycle

Then the following Boolean expression defines C_5:

$$C_5 = \overline{P} \bullet \overline{Q} \bullet T_2 + \overline{P} \bullet Q \bullet T_2$$

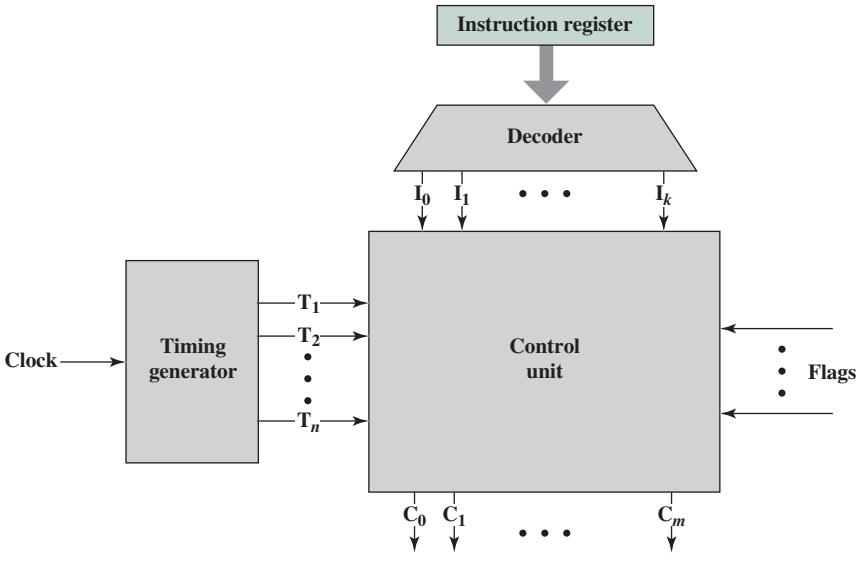


Figure 20.10 Control Unit with Decoded Inputs

That is, the control signal C_5 will be asserted during the second time unit of both the fetch and indirect cycles.

This expression is not complete. C_5 is also needed during the execute cycle. For our simple example, let us assume that there are only three instructions that read from memory: LDA, ADD, and AND. Now we can define C_5 as

$$C_5 = \overline{P}\bullet\overline{Q}\bullet T_2 + \overline{P}\bullet Q\bullet T_2 + P\bullet\overline{Q}\bullet(\text{LDA} + \text{ADD} + \text{AND})\bullet T_2$$

This same process could be repeated for every control signal generated by the processor. The result would be a set of Boolean equations that define the behavior of the control unit and hence of the processor.

To tie everything together, the control unit must control the state of the instruction cycle. As was mentioned, at the end of each subcycle (fetch, indirect, execute, interrupt), the control unit issues a signal that causes the timing generator to reinitialize and issue T_1. The control unit must also set the appropriate values of P and Q to define the next subcycle to be performed.

The reader should be able to appreciate that in a modern complex processor, the number of Boolean equations needed to define the control unit is very large. The task of implementing a combinatorial circuit that satisfies all of these equations becomes extremely difficult. The result is that a far simpler approach, known as *microprogramming*, is usually used. This is the subject of the next chapter.

20.4 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

control bus	control signal	hardwired implementation
control path	control unit	micro-operations

Review Questions

20.1 Explain the distinction between the written sequence and the time sequence of an instruction.
20.2 What is the relationship between instructions and micro-operations?
20.3 What is the overall function of a processor's control unit?
20.4 Outline a three-step process that leads to a characterization of the control unit.
20.5 What basic tasks does a control unit perform?
20.6 Provide a typical list of the inputs and outputs of a control unit.
20.7 List three types of control signals.
20.8 Briefly explain what is meant by a hardwired implementation of a control unit.

Problems

20.1 Your ALU can add its two input registers, and it can logically complement the bits of either input register, but it cannot subtract. Numbers are to be stored in twos complement representation. List the micro-operations your control unit must perform to cause a subtraction.

20.2 Show the micro-operations and control signals in the same fashion as Table 20.1 for the processor in Figure 20.5 for the following instructions:
- Load Accumulator
- Store Accumulator
- Add to Accumulator
- AND to Accumulator
- Jump
- Jump if AC = 0
- Complement Accumulator

20.3 Assume that propagation delay along the bus and through the ALU of Figure 20.6 are 20 and 100 ns, respectively. The time required for a register to copy data from the bus is 10 ns. What is the time that must be allowed for
a. data from one register to another?
b. the program counter?

20.4 Write the sequence of micro-operations required for the bus structure of Figure 20.6 to add a number to the AC when the number is
a. immediate operand;
b. direct-address operand;
c. indirect-address operand.

20.5 A stack is implemented as shown in Figure 20.11 (see Appendix I for a discussion of stacks). Show the sequence of micro-operations for
a. popping;
b. the stack.

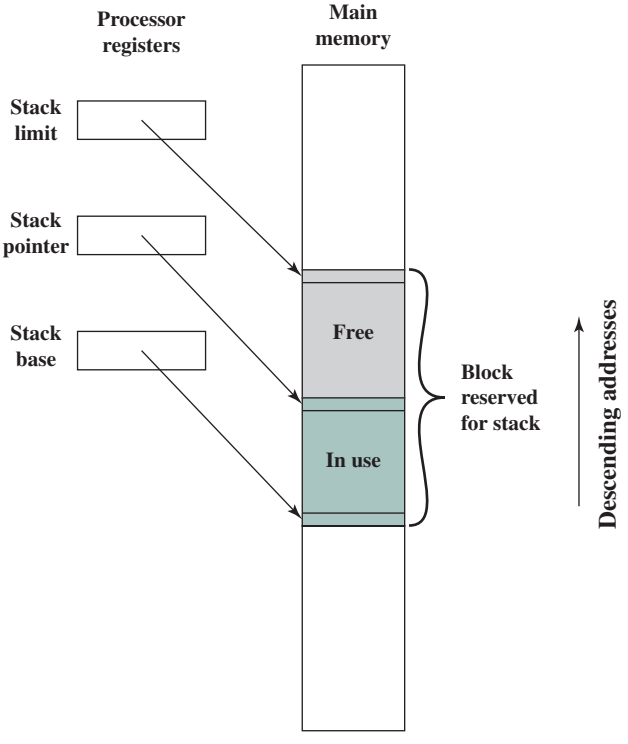


Figure 20.11 Typical Stack Organization (full/descending)

CHAPTER 21

MICROPROGRAMMED CONTROL

21.1 Basic Concepts
- Microinstructions
- Microprogrammed Control Unit
- Wilkes Control
- Advantages and Disadvantages

21.2 Microinstruction Sequencing
- Design Considerations
- Sequencing Techniques
- Address Generation
- LSI-11 Microinstruction Sequencing

21.3 Microinstruction Execution
- A Taxonomy of Microinstructions
- Microinstruction Encoding
- LSI-11 Microinstruction Execution
- IBM 3033 Microinstruction Execution

21.4 TI 8800
- Microinstruction Format
- Microsequencer
- Registered ALU

21.5 Key Terms, Review Questions, and Problems

Learning Objectives

After studying this chapter, you should be able to:

- Present an overview of the basic concepts of microprogrammed control.
- Understand the difference between hardwired control and microprogrammed control.
- Discuss the basic categories of sequencing techniques.
- Present an overview of the taxonomy of microinstructions.

The term *microprogram* was first coined by M. V. Wilkes in the early 1950s [WILK51]. Wilkes proposed an approach to control unit design that was organized and systematic and avoided the complexities of a hardwired implementation. The idea intrigued many researchers but appeared unworkable because it would require a fast, relatively inexpensive control memory.

The state of the microprogramming art was reviewed by *Datamation* in its February 1964 issue. No microprogrammed system was in wide use at that time, and one of the papers [HILL64] summarized the then-popular view that the future of microprogramming "is somewhat cloudy. None of the major manufacturers has evidenced interest in the technique, although presumably all have examined it."

This situation changed dramatically within a very few months. IBM's System/360 was announced in April, and all but the largest models were microprogrammed. Although the 360 series predated the availability of semiconductor ROM, the advantages of microprogramming were compelling enough for IBM to make this move. Microprogramming became a popular technique for implementing the control unit of CISC processors. In recent years, microprogramming has become less used but remains a tool available to computer designers. For example, as we have seen on the Pentium 4, machine instructions are converted into a RISC-like format, most of which are executed without the use of microprogramming. However, some of the instructions are executed using microprogramming.

21.1 BASIC CONCEPTS

Microinstructions

The control unit seems a reasonably simple device. Nevertheless, to implement a control unit as an interconnection of basic logic elements is no easy task. The design must include logic for sequencing through micro-operations, for executing micro-operations, for interpreting opcodes, and for making decisions based on ALU flags. It is difficult to design and test such a piece of hardware. Furthermore, the design is relatively inflexible. For example, it is difficult to change the design if one wishes to add a new machine instruction.

An alternative, which has been used in many CISC processors, is to implement a **microprogrammed control unit**.

Consider Table 21.1. In addition to the use of control signals, each micro-operation is described in symbolic notation. This notation looks suspiciously like a programming language. In fact it is a language, known as a **microprogramming language**. Each line describes a set of micro-operations occurring at one time and is known as a **microinstruction**. A sequence of instructions is known as a **microprogram**, or *firmware*. This latter term reflects the fact that a microprogram is midway between hardware and software. It is easier to design in firmware than hardware, but it is more difficult to write a firmware program than a software program.

How can we use the concept of microprogramming to implement a control unit? Consider that for each micro-operation, all that the control unit is allowed to do is generate a set of control signals. Thus, for any micro-operation, each control line emanating from the control unit is either on or off. This condition can, of course, be represented by a binary digit for each control line. So we could construct a *control word* in which each bit represents one control line. Then each micro-operation would be represented by a different pattern of 1s and 0s in the control word.

Suppose we string together a sequence of control words to represent the sequence of micro-operations performed by the control unit. Next, we must recognize that the sequence of micro-operations is not fixed. Sometimes we have an indirect cycle; sometimes we do not. So let us put our control words in a memory, with each word having a unique address. Now add an address field to each control word, indicating the location of the next control word to be executed if a certain condition is true (e.g., the indirect bit in a memory-reference instruction is 1). Also, add a few bits to specify the condition.

Table 21.1 Machine Instruction Set for Wilkes Example

Order	Effect of Order
$A\ n$	$C(Acc) + C(n)$ to Acc_1
$S\ n$	$C(Acc) - C(n)$ to Acc_1
$H\ n$	$C(n)$ to Acc_2
$V\ n$	$C(Acc2) \times C(n)$ to Acc, where $C(n) \geq 0$
$T\ n$	$C(Acc_1)$ to n, 0 to Acc
$U\ n$	$C(Acc_1)$ to n
$R\ n$	$C(Acc) \times 2^{(n+1)}$ to Acc
$L\ n$	$C(Acc) \times 2^{n+1}$ to Acc
$G\ n$	IF $C(Acc) < 0$, transfer control to n; if $C(Acc) \geq 0$, ignore (i.e., proceed serially)
$I\ n$	Read next character on input mechanism into n
$O\ n$	Send $C(n)$ to output mechanism

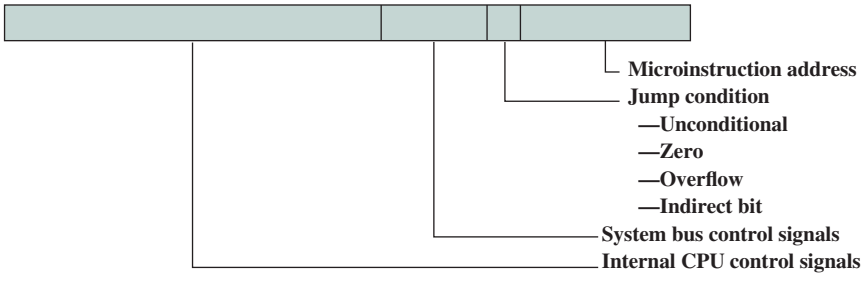
Notation: Acc = accumulator
Acc_1 = most significant half of accumulator
Acc_2 = least significant half of accumulator
n = storage location n
$C(X)$ = contents of X (X = register or storage location)

The result is known as a **horizontal microinstruction**, an example of which is shown in Figure 21.1a. The format of the microinstruction or control word is as follows. There is one bit for each internal processor control line and one bit for each system bus control line. There is a condition field indicating the condition under which there should be a branch, and there is a field with the address of the microinstruction to be executed next when a branch is taken. Such a microinstruction is interpreted as follows:

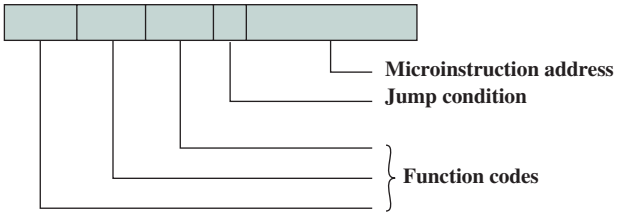
1. To execute this microinstruction, turn on all the control lines indicated by a 1 bit; leave off all control lines indicated by a 0 bit. The resulting control signals will cause one or more micro-operations to be performed.
2. If the condition indicated by the condition bits is false, execute the next microinstruction in sequence.
3. If the condition indicated by the condition bits is true, the next microinstruction to be executed is indicated in the address field.

Figure 21.2 shows how these control words or microinstructions could be arranged in a **control memory**. The microinstructions in each routine are to be executed sequentially. Each routine ends with a branch or jump instruction indicating where to go next. There is a special execute cycle routine whose only purpose is to signify that one of the machine instruction routines (AND, ADD, and so on) is to be executed next, depending on the current opcode.

The control memory of Figure 21.2 is a concise description of the complete operation of the control unit. It defines the sequence of micro-operations to be



(a) Horizontal microinstruction



(b) Vertical microinstruction

Figure 21.1 Typical Microinstruction Formats

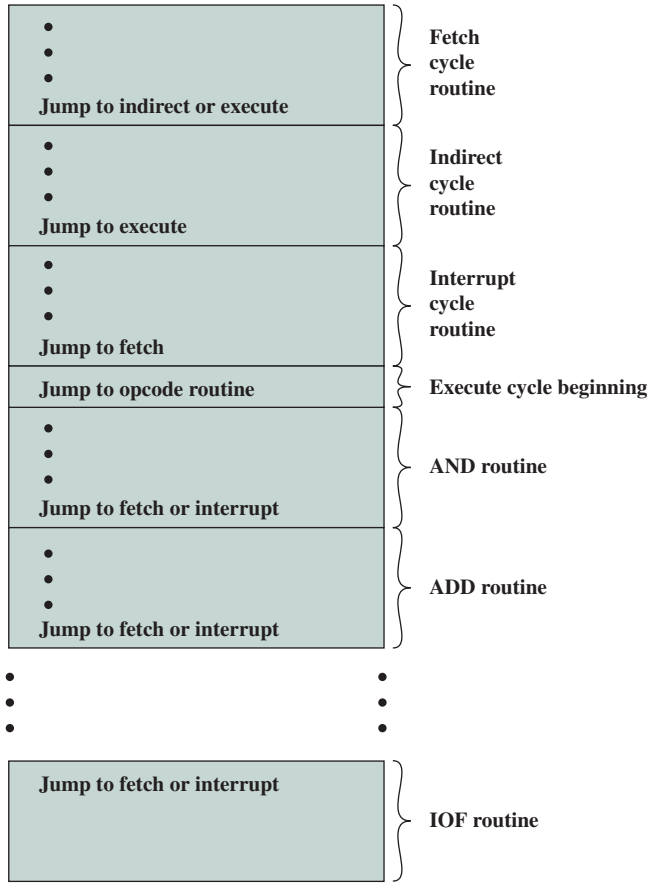


Figure 21.2 Organization of Control Memory

performed during each cycle (fetch, indirect, execute, interrupt), and it specifies the sequencing of these cycles. If nothing else, this notation would be a useful device for documenting the functioning of a control unit for a particular computer. But it is more than that. It is also a way of implementing the control unit.

Microprogrammed Control Unit

The control memory of Figure 21.2 contains a program that describes the behavior of the control unit. It follows that we could implement the control unit by simply executing that program.

Figure 21.3 shows the key elements of such an implementation. The set of microinstructions is stored in the *control memory*. The *control address register* contains the address of the next microinstruction to be read. When a microinstruction is read from the control memory, it is transferred to a *control buffer register*. The left-hand portion of that register (see Figure 21.1a) connects to the control lines emanating from the control unit. Thus, *reading* a microinstruction from the control memory is the same as *executing* that microinstruction. The third element shown in the figure is a sequencing unit that loads the control address register and issues a read command.

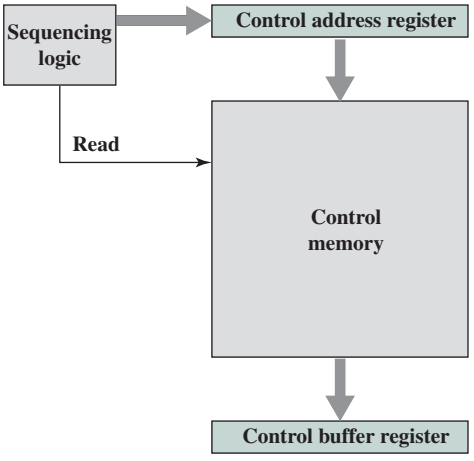


Figure 21.3 Control Unit Microarchitecture

Let us examine this structure in greater detail, as depicted in Figure 21.4. Comparing this with Figure 21.3, we see that the control unit still has the same inputs (IR, ALU flags, clock) and outputs (control signals). The control unit functions as follows:

1. To execute an instruction, the sequencing logic unit issues a READ command to the control memory.
2. The word whose address is specified in the control address register is read into the control buffer register.
3. The content of the control buffer register generates control signals and next-address information for the sequencing logic unit.
4. The sequencing logic unit loads a new address into the control address register based on the next-address information from the control buffer register and the ALU flags.

All this happens during one clock pulse.

The last step just listed needs elaboration. At the conclusion of each microinstruction, the sequencing logic unit loads a new address into the control address register. Depending on the value of the ALU flags and the control buffer register, one of three decisions is made:

- **Get the next instruction:** Add 1 to the control address register.
- **Jump to a new routine based on a jump microinstruction:** Load the address field of the control buffer register into the control address register.
- **Jump to a machine instruction routine:** Load the control address register based on the opcode in the IR.

Figure 21.4 shows two modules labeled *decoder*. The upper decoder translates the opcode of the IR into a control memory address. The lower decoder is not used for horizontal microinstructions but is used for **vertical microinstructions** (Figure 21.1b). As was mentioned, in a horizontal microinstruction every bit in the

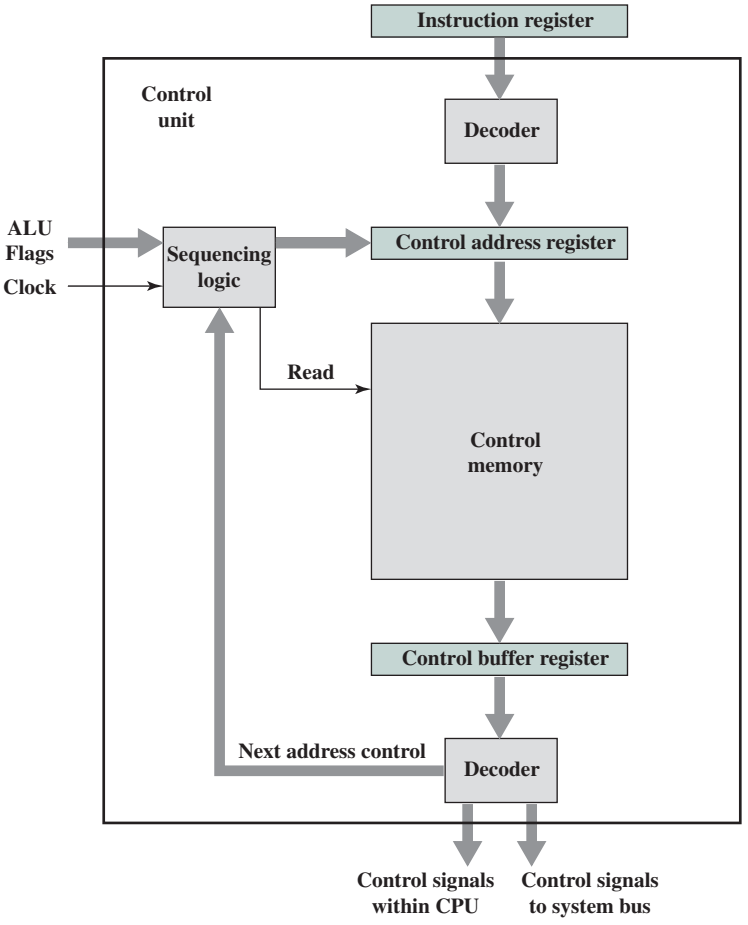


Figure 21.4 Functioning of Microprogrammed Control Unit

control field attaches to a control line. In a vertical microinstruction, a code is used for each action to be performed [e.g., MAR ← (PC)], and the decoder translates this code into individual control signals. The advantage of vertical microinstructions is that they are more compact (fewer bits) than horizontal microinstructions, at the expense of a small additional amount of logic and time delay.

Wilkes Control

As was mentioned, Wilkes first proposed the use of a microprogrammed control unit in 1951 [WILK51]. This proposal was subsequently elaborated into a more detailed design [WILK53]. It is instructive to examine this seminal proposal.

The configuration proposed by Wilkes is depicted in Figure 21.5. The heart of the system is a matrix partially filled with diodes. During a machine cycle, one row of the matrix is activated with a pulse. This generates signals at those points where a diode is present (indicated by a dot in the diagram). The first part of the row generates the control signals that control the operation of the processor. The second part generates the

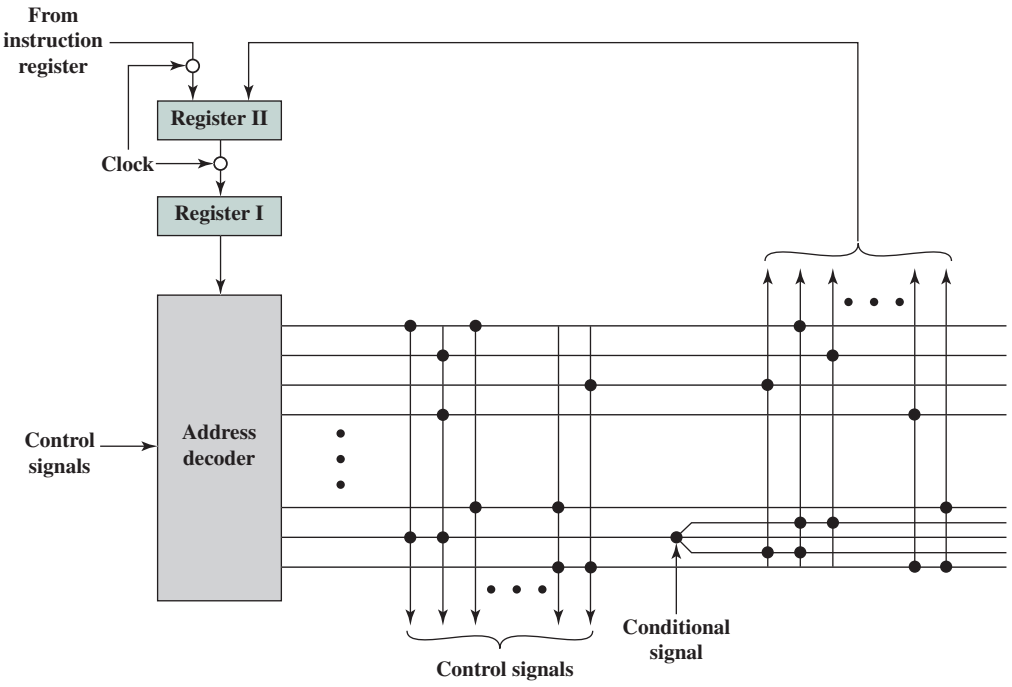


Figure 21.5 Wilkes's Microprogrammed Control Unit

address of the row to be pulsed in the next machine cycle. Thus, each row of the matrix is one microinstruction, and the layout of the matrix is the control memory.

At the beginning of the cycle, the address of the row to be pulsed is contained in Register I. This address is the input to the decoder, which, when activated by a clock pulse, activates one row of the matrix. Depending on the control signals, either the opcode in the instruction register or the second part of the pulsed row is passed into Register II during the cycle. Register II is then gated to Register I by a clock pulse. Alternating clock pulses are used to activate a row of the matrix and to transfer from Register II to Register I. The two-register arrangement is needed because the decoder is simply a combinatorial circuit; with only one register, the output would become the input during a cycle, causing an unstable condition.

This scheme is very similar to the horizontal microprogramming approach described earlier (Figure 21.1a). The main difference is this: In the previous description, the control address register could be incremented by one to get the next address. In the Wilkes scheme, the next address is contained in the microinstruction. To permit branching, a row must contain two address parts, controlled by a conditional signal (e.g., flag), as shown in the figure.

Having proposed this scheme, Wilkes provides an example of its use to implement the control unit of a simple machine. This example, the first known design of a microprogrammed processor, is worth repeating here because it illustrates many of the contemporary principles of microprogramming.

The processor of the hypothetical machine (the example machine by Wilkes) includes the following registers:

A	Multiplicand
B	Accumulator (least significant half)
C	Accumulator (most significant half)
D	Shift register

In addition, there are three registers and two 1-bit flags accessible only to the control unit. The registers are as follows:

E	Serves as both a memory address register (MAR) and temporary storage
F	Program counter
G	Another temporary register; used for counting

Table 21.1 lists the machine instruction set for this example. Table 21.2 is the complete set of microinstructions, expressed in symbolic form, that implements the control unit. Thus, a total of 38 microinstructions is all that is required to define the system completely.

The first full column gives the address (row number) of each microinstruction. Those addresses corresponding to opcodes are labeled. Thus, when the opcode for the add instruction (A) is encountered, the microinstruction at location 5 is executed. Columns 2 and 3 express the actions to be taken by the ALU and control unit, respectively. Each symbolic expression must be translated into a set of control signals (microinstruction bits). Columns 4 and 5 have to do with the setting and use of the two flags (flip-flops). Column 4 specifies the signal that sets the flag. For example, $(1)C_s$ means that flag number 1 is set by the sign bit of the number in register C. If column 5 contains a flag identifier, then columns 6 and 7 contain the two alternative microinstruction addresses to be used. Otherwise, column 6 specifies the address of the next microinstruction to be fetched.

Instructions 0 through 4 constitute the fetch cycle. Microinstruction 4 presents the opcode to a decoder, which generates the address of a microinstruction corresponding to the machine instruction to be fetched. The reader should be able to deduce the complete functioning of the control unit from a careful study of Table 21.2.

Advantages and Disadvantages

The principal advantage of the use of microprogramming to implement a control unit is that it simplifies the design of the control unit. Thus, it is both cheaper and less error prone to implement. A *hardwired* control unit must contain complex logic for sequencing through the many micro-operations of the instruction cycle. On the other hand, the decoders and sequencing logic unit of a microprogrammed control unit are very simple pieces of logic.

The principal disadvantage of a microprogrammed unit is that it will be somewhat slower than a hardwired unit of comparable technology. Despite this, microprogramming is the dominant technique for implementing control units in pure CISC architectures, due to its ease of implementation. RISC processors, with their simpler instruction format, typically use hardwired control units. We now examine the microprogrammed approach in greater detail.

Table 21.2 Microinstructions for Wilkes Example

Notations: $A, B, C, \ldots$ stand for the various registers in the arithmetical and control register units. C to D indicates that the switching circuits connect the output of register C to the input register D; $(D + A)$ to C indicates that the output register of A is connected to the one input of the adding unit (the output of D is permanently connected to the other input), and the output of the adder to register C. A numerical symbol n in quotes (e.g., "n") stands for the source whose output is the number n in units of the least significant digit.

		Arithmetical Unit	Control Register Unit	Conditional Flip-Flop		Next Microinstruction	
				Set	Use	0	1
	0		*F* to *G* and *E*			1	
	1		(*G* to "1") to *F*			2	
	2		Store to *G*			3	
	3		*G* to *E*			4	
	4		*E* to decoder			—	
A	5	*C* to *D*				16	
S	6	*C* to *D*				17	
H	7	Store to *B*				0	
V	8	Store to *A*				27	
T	9	*C* to Store				25	
U	10	*C* to Store				0	
R	11	*B* to *D*	*E* to *G*			19	
L	12	*C* to *D*	*E* to *G*			22	
G	13		*E* to *G*	(1)C_5		18	
I	14	Input to Store				0	
O	15	Store to Output				0	
	16	(*D* + Store) to *C*				0	
	17	(*D* − Store) to *C*				0	
	18				1	0	1
	19	*D* to *B* (*R*)*	(*G* − 1) to *E*			20	
	20	*C* to *D*		(1)E_5		21	
	21	*D* to *C* (*R*)			1	11	0
	22	*D* to *C* (*L*)†	(*G* − 1) to *E*			23	
	23	*B* to *D*		(1)E_5		24	
	24	*D* to *B* (*L*)			1	12	0
	25	"0" to *B*				26	
	26	*B* to *C*				0	
	27	"0" to *C*	"18" to *E*			28	
	28	*B* to *D*	*E* to *G*	(1)B_1		29	
	29	*D* to *B* (*R*)	(*G* − "1") to *E*			30	
	30	*C* to *D* (*R*)		(2)E_5	1	31	32

		Arithmetical Unit	Control Register Unit	Conditional Flip-Flop		Next Microinstruction	
				Set	Use	0	1
	31	D to C			2	28	33
	32	$(D + A)$ to C			2	28	33
	33	B to D		$(1)B_1$		34	
	34	D to B (R)				35	
	35	C to D (R)			1	36	37
	36	D to C				0	
	37	$(D - A)$ to C				0	

* Right shift. The switching circuits in the arithmetic unit are arranged so that the least significant digit of the register C is placed in the most significant place of register B during right shift micro-operations, and the most significant digit of register C (sign digit) is repeated (thus making the correction for negative numbers).

† Left shift. The switching circuits are similarly arranged to pass the most significant digit of register B to the least significant place of register C during left shift micro-operations.

21.2 MICROINSTRUCTION SEQUENCING

The two basic tasks performed by a microprogrammed control unit are as follows:

- **Microinstruction sequencing:** Get the next microinstruction from the control memory.
- **Microinstruction execution:** Generate the control signals needed to execute the microinstruction.

In designing a control unit, these tasks must be considered together, because both affect the format of the microinstruction and the timing of the control unit. In this section, we will focus on sequencing and say as little as possible about format and timing issues. These issues are examined in more detail in the next section.

Design Considerations

Two concerns are involved in the design of a microinstruction sequencing technique: the size of the microinstruction and the address-generation time. The first concern is obvious; minimizing the size of the control memory reduces the cost of that component. The second concern is simply a desire to execute microinstructions as fast as possible.

In executing a microprogram, the address of the next microinstruction to be executed is in one of these categories:

- Determined by instruction register
- Next sequential address
- Branch

The first category occurs only once per instruction cycle, just after an instruction is fetched. The second category is the most common in most designs. However, the design cannot be optimized just for sequential access. Branches, both conditional and unconditional, are a necessary part of a microprogram. Furthermore, microinstruction sequences tend to be short; one out of every three or four microinstructions could be a branch [SIEW82]. Thus, it is important to design compact, time-efficient techniques for microinstruction branching.

Sequencing Techniques

Based on the current microinstruction, condition flags, and the contents of the instruction register, a control memory address must be generated for the next microinstruction. A wide variety of techniques have been used. We can group them into three general categories, as illustrated in Figures 21.6 to 21.8. These categories are based on the format of the address information in the microinstruction:

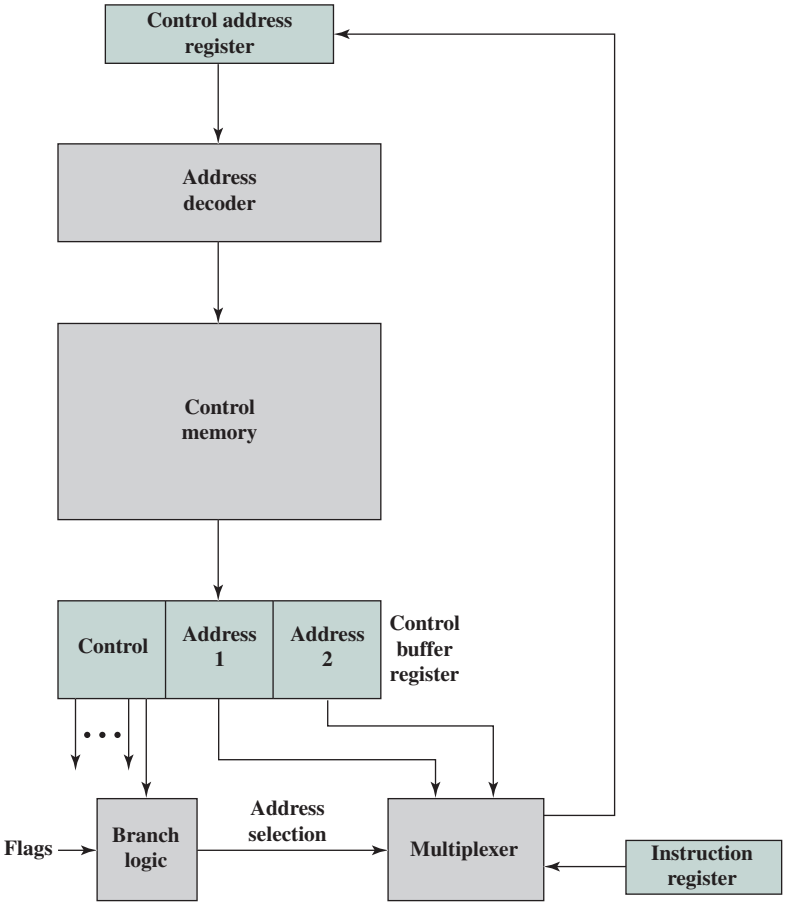


Figure 21.6 Branch Control Logic: Two Address Fields

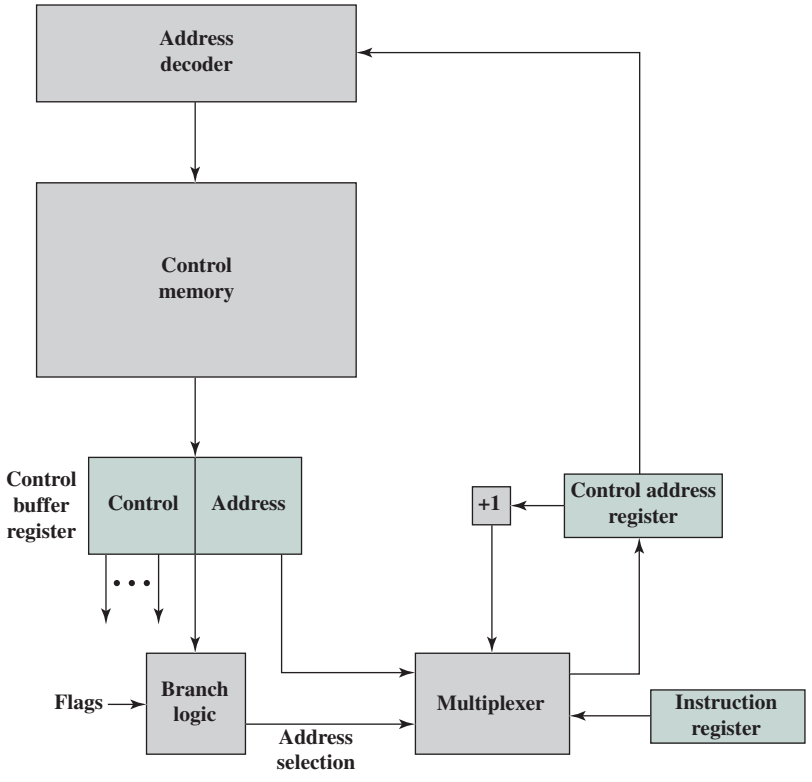


Figure 21.7 Branch Control Logic: Single Address Field

- Two address fields
- Single address field
- Variable format

The simplest approach is to provide two address fields in each microinstruction. Figure 21.6 suggests how this information is to be used. A multiplexer is provided that serves as a destination for both address fields plus the instruction register. Based on an address-selection input, the multiplexer transmits either the opcode or one of the two addresses to the control address register (CAR). The CAR is subsequently decoded to produce the next microinstruction address. The address-selection signals are provided by a branch logic module whose input consists of control unit flags plus bits from the control portion of the microinstruction.

Although the two-address approach is simple, it requires more bits in the microinstruction than other approaches. With some additional logic, savings can be achieved. A common approach is to have a single address field (Figure 21.7). With this approach, the options for next address are as follows:

- Address field
- Instruction register code
- Next sequential address

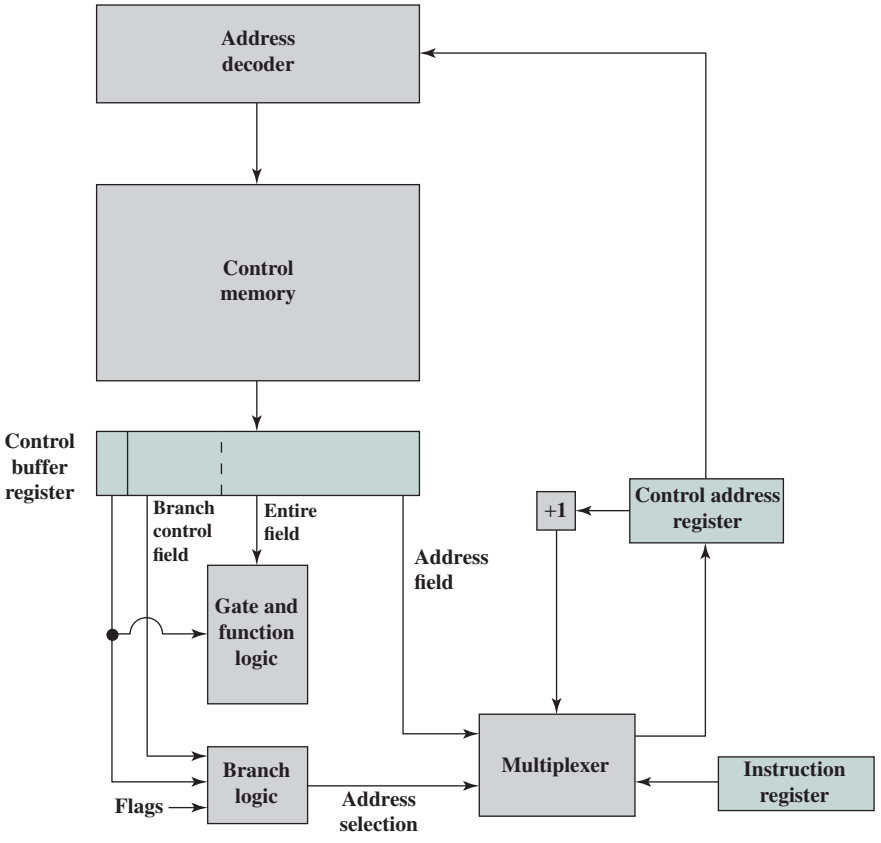


Figure 21.8 Branch Control Logic: Variable Format

The address-selection signals determine which option is selected. This approach reduces the number of address fields to one. Note, however, that the address field often will not be used. Thus, there is some inefficiency in the microinstruction coding scheme.

Another approach is to provide for two entirely different microinstruction formats (Figure 21.8). One bit designates which format is being used. In one format, the remaining bits are used to activate control signals. In the other format, some bits drive the branch logic module, and the remaining bits provide the address. With the first format, the next address is either the next sequential address or an address derived from the instruction register. With the second format, either a conditional or unconditional branch is being specified. One disadvantage of this approach is that one entire cycle is consumed with each branch microinstruction. With the other approaches, address generation occurs as part of the same cycle as control signal generation, minimizing control memory accesses.

The approaches just described are general. Specific implementations will often involve a variation or combination of these techniques.

Address Generation

We have looked at the sequencing problem from the point of view of format considerations and general logic requirements. Another viewpoint is to consider the various ways in which the next address can be derived or computed.

Table 21.3 lists the various address generation techniques. These can be divided into explicit techniques, in which the address is explicitly available in the microinstruction, and implicit techniques, which require additional logic to generate the address.

We have essentially dealt with the explicit techniques. With a two-field approach, two alternative addresses are available with each microinstruction. Using either a single address field or a variable format, various branch instructions can be implemented. A conditional branch instruction depends on the following types of information:

- ALU flags
- Part of the opcode or address mode fields of the machine instruction
- Parts of a selected register, such as the sign bit
- Status bits within the control unit

Several implicit techniques are also commonly used. One of these, mapping, is required with virtually all designs. The opcode portion of a machine instruction must be mapped into a microinstruction address. This occurs only once per instruction cycle.

A common implicit technique is one that involves combining or adding two portions of an address to form the complete address. This approach was taken for the IBM S/360 family [TUCK67] and used on many of the S/370 models. We will use the IBM 3033 as an example.

The control address register on the IBM 3033 is 13 bits long and is illustrated in Figure 21.9. Two parts of the address can be distinguished. The highest-order 8 bits (00–07) normally do not change from one microinstruction cycle to the next. During the execution of a microinstruction, these 8 bits are copied directly from an 8-bit field of the microinstruction (the BA field) into the highest-order 8 bits of the control address register. This defines a block of 32 microinstructions in control memory. The remaining 5 bits of the control address register are set to specify the specific address of the microinstruction to be fetched next. Each of these bits is determined by a 4-bit field (except one is a 7-bit field) in the current microinstruction; the field specifies the condition for setting the corresponding bit. For example, a bit in the control address register might be set to 1 or 0 depending on whether a carry occurred on the last ALU operation.

Table 21.3 Microinstruction Address Generation Techniques

Explicit	Implicit
Two-field	Mapping
Unconditional branch	Addition
Conditional branch	Residual control

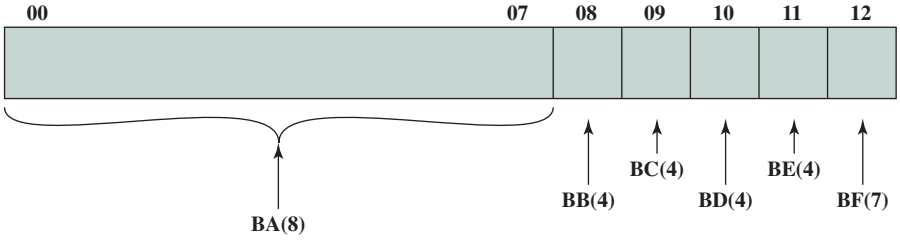


Figure 21.9 IBM 3033 Control Address Register

The final approach listed in Table 21.3 is termed *residual control*. This approach involves the use of a microinstruction address that has previously been saved in temporary storage within the control unit. For example, some microinstruction sets come equipped with a subroutine facility. An internal register or stack of registers is used to hold return addresses. An example of this approach is taken on the LSI-11, which we now examine.

LSI-11 Microinstruction Sequencing

The LSI-11 is a microcomputer version of a PDP-11, with the main components of the system residing on a single board. The LSI-11 is implemented using a microprogrammed control unit [SEBE76].

The LSI-11 makes use of a 22-bit microinstruction and a control memory of 2K 22-bit words. The next microinstruction address is determined in one of five ways:

- **Next sequential address:** In the absence of other instructions, the control unit's control address register is incremented by 1.
- **Opcode mapping:** At the beginning of each instruction cycle, the next microinstruction address is determined by the opcode.
- **Subroutine facility:** Explained presently.
- **Interrupt testing:** Certain microinstructions specify a test for interrupts. If an interrupt has occurred, this determines the next microinstruction address.
- **Branch:** Conditional and unconditional branch microinstructions are used.

A one-level subroutine facility is provided. One bit in every microinstruction is dedicated to this task. When the bit is set, an 11-bit return register is loaded with the updated contents of the control address register. A subsequent microinstruction that specifies a return will cause the control address register to be loaded from the return register.

The return is one form of unconditional branch instruction. Another form of unconditional branch causes the bits of the control address register to be loaded from 11 bits of the microinstruction. The conditional branch instruction makes use of a 4-bit test code within the microinstruction. This code specifies testing of various ALU condition codes to determine the branch decision. If the condition is not true, the next sequential address is selected. If it is true, the 8 lowest-order bits of the control address register are loaded from 8 bits of the microinstruction. This allows branching within a 256-word page of memory.

As can be seen, the LSI-11 includes a powerful address sequencing facility within the control unit. This allows the microprogrammer considerable flexibility and can ease the microprogramming task. On the other hand, this approach requires more control unit logic than do simpler capabilities.

21.3 MICROINSTRUCTION EXECUTION

The microinstruction cycle is the basic event on a microprogrammed processor. Each cycle is made up of two parts: fetch and execute. The fetch portion is determined by the generation of a microinstruction address, and this was dealt with in the preceding section. This section deals with the execution of a microinstruction.

Recall that the effect of the execution of a microinstruction is to generate control signals. Some of these signals control points internal to the processor. The remaining signals go to the external control bus or other external interface. As an incidental function, the address of the next microinstruction is determined.

The preceding description suggests the organization of a control unit shown in Figure 21.10. This slightly revised version of Figure 21.4 emphasizes the focus of this section. The major modules in this diagram should by now be clear. The sequencing logic module contains the logic to perform the functions discussed in the preceding section. It generates the address of the next microinstruction, using as inputs the instruction register, ALU flags, the control address register (for incrementing), and the control buffer register. The last may provide an actual address, control bits, or both. The module is driven by a clock that determines the timing of the microinstruction cycle.

The control logic module generates control signals as a function of some of the bits in the microinstruction. It should be clear that the format and content of the microinstruction determines the complexity of the control logic module.

A Taxonomy of Microinstructions

Microinstructions can be classified in a variety of ways. Distinctions that are commonly made in the literature include the following:

- Vertical/horizontal
- Packed/unpacked
- Hard/soft microprogramming
- Direct/indirect encoding

All of these bear on the format of the microinstruction. None of these terms has been used in a consistent, precise way in the literature. However, an examination of these pairs of qualities serves to illuminate microinstruction design alternatives. In the following paragraphs, we first look at the key design issue underlying all of these pairs of characteristics, and then we look at the concepts suggested by each pair.

In the original proposal by Wilkes [WILK51], each bit of a microinstruction either directly produced a control signal or directly produced one bit of the next address. We have seen, in the preceding section, that more complex address

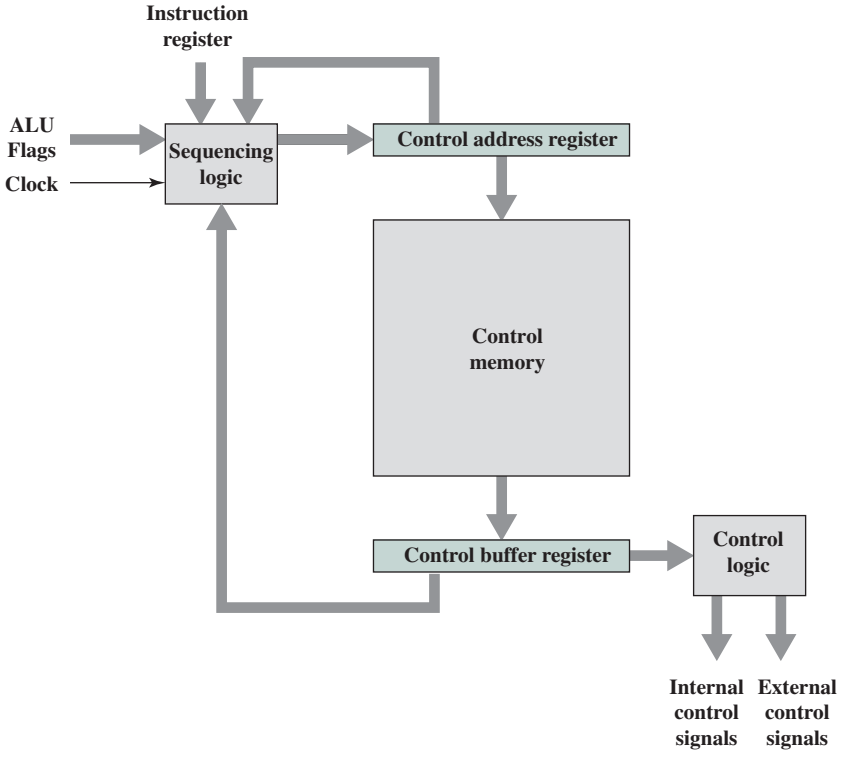


Figure 21.10 Control Unit Organization

sequencing schemes, using fewer microinstruction bits, are possible. These schemes require a more complex sequencing logic module. A similar sort of trade-off exists for the portion of the microinstruction concerned with control signals. By encoding control information, and subsequently decoding it to produce control signals, control word bits can be saved.

How can this encoding be done? To answer that, consider that there are a total of K different internal and external control signals to be driven by the control unit. In Wilkes's scheme, K bits of the microinstruction would be dedicated to this purpose. This allows all of the $2K$ possible combinations of control signals to be generated during any instruction cycle. But we can do better than this if we observe that not all of the possible combinations will be used. Examples include the following:

- Two sources cannot be gated to the same destination (e.g., C_2 and C_8 in Figure 21.5).
- A register cannot be both source and destination (e.g., C_5 and C_{12} in Figure 21.5).
- Only one pattern of control signals can be presented to the ALU at a time.
- Only one pattern of control signals can be presented to the external control bus at a time.

So, for a given processor, all possible allowable combinations of control signals could be listed, giving some number $Q < 2^K$ possibilities. These could be encoded with a minimum $\log_2 Q$ bits, with $(\log_2 Q) < K$. This would be the tightest possible form of encoding that preserves all allowable combinations of control signals. In practice, this form of encoding is not used, for two reasons:

- It is as difficult to program as a pure decoded (Wilkes) scheme. This point is discussed further presently.
- It requires a complex and therefore slow control logic module.

Instead, some compromises are made. These are of two kinds:

- More bits than are strictly necessary are used to encode the possible combinations.
- Some combinations that are physically allowable are not possible to encode.

The latter kind of compromise has the effect of reducing the number of bits. The net result, however, is to use more than $\log_2 Q$ bits.

In the next subsection, we will discuss specific encoding techniques. The remainder of this subsection deals with the effects of encoding and the various terms used to describe it.

Based on the preceding, we can see that the control signal portion of the microinstruction format falls on a spectrum. At one extreme, there is one bit for each control signal; at the other extreme, a highly encoded format is used. Table 21.4 shows that other characteristics of a microprogrammed control unit also fall along a spectrum and that these spectra are, by and large, determined by the degree-of-encoding spectrum.

The second pair of items in the table is rather obvious. The pure Wilkes scheme will require the most bits. It should also be apparent that this extreme presents the most detailed view of the hardware. Every control signal is individually controllable

Table 21.4 The Microinstruction Spectrum

Characteristics	
Unencoded	Highly encoded
Many bits	Few bits
Detailed view of hardware	Aggregated view of hardware
Difficult to program	Easy to program
Concurrency fully exploited	Concurrency not fully exploited
Little or no control logic	Complex control logic
Fast execution	Slow execution
Optimize performance	Optimize programming
Terminology	
Unpacked	Packed
Horizontal	Vertical
Hard	Soft

by the microprogrammer. Encoding is done in such a way as to aggregate functions or resources, so that the microprogrammer is viewing the processor at a higher, less detailed level. Furthermore, the encoding is designed to ease the microprogramming burden. Again, it should be clear that the task of understanding and orchestrating the use of all the control signals is a difficult one. As was mentioned, one of the consequences of encoding, typically, is to prevent the use of certain otherwise allowable combinations.

The preceding paragraph discusses microinstruction design from the microprogrammer's point of view. But the degree of encoding also can be viewed from its hardware effects. With a pure unencoded format, little or no decode logic is needed; each bit generates a particular control signal. As more compact and more aggregated encoding schemes are used, more complex decode logic is needed. This, in turn, may affect performance. More time is needed to propagate signals through the gates of the more complex control logic module. Thus, the execution of encoded microinstructions takes longer than the execution of unencoded ones.

Therefore, all of the characteristics listed in Table 21.4 fall along a spectrum of design strategies. In general, a design that falls toward the left end of the spectrum is intended to optimize the performance of the control unit. Designs toward the right end are more concerned with optimizing the process of microprogramming. Indeed, microinstruction sets near the right end of the spectrum look very much like machine instruction sets. A good example of this is the LSI-11 design, described later in this section. Typically, when the objective is simply to implement a control unit, the design will be near the left end of the spectrum. The IBM 3033 design, discussed presently, is in this category. As we shall discuss later, some systems permit a variety of users to construct different microprograms using the same microinstruction facility. In the latter cases, the design is likely to fall near the right end of the spectrum.

We can now deal with some of the terminology introduced earlier. Table 21.4 indicates how three of these pairs of terms relate to the microinstruction spectrum. In essence, all of these pairs describe the same thing but emphasize different design characteristics.

The degree of packing relates to the degree of identification between a given control task and specific microinstruction bits. As the bits become more *packed*, a given number of bits contains more information. Thus, packing connotes encoding. The terms *horizontal* and *vertical* relate to the relative width of microinstructions. [SIEW82] suggests as a rule of thumb that vertical microinstructions have lengths in the range of 16 to 40 bits and that horizontal microinstructions have lengths in the range of 40 to 100 bits. The terms *hard* and *soft* microprogramming are used to suggest the degree of closeness to the underlying control signals and hardware layout. Hard microprograms are generally fixed and committed to read-only memory. Soft microprograms are more changeable and are suggestive of user microprogramming.

The other pair of terms mentioned at the beginning of this subsection refers to direct versus indirect encoding, a subject to which we now turn.

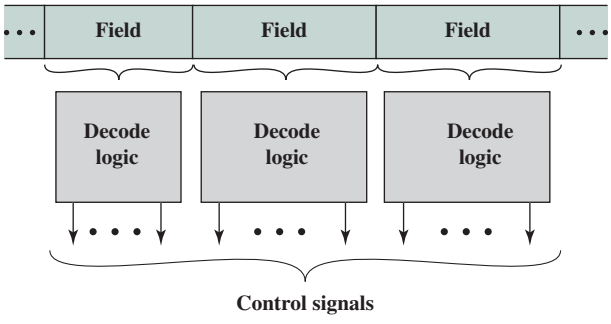
Microinstruction Encoding

In practice, microprogrammed control units are not designed using a pure unencoded or horizontal microinstruction format. At least some degree of encoding is used to reduce control memory width and to simplify the task of microprogramming.

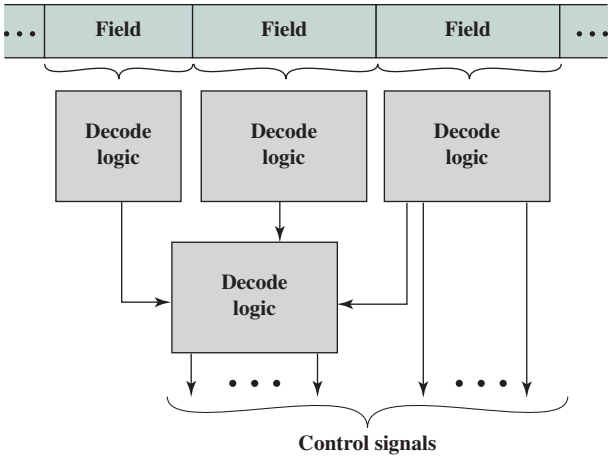
The basic technique for encoding is illustrated in Figure 21.11a. The microinstruction is organized as a set of fields. Each field contains a code, which, upon decoding, activates one or more control signals.

Let us consider the implications of this layout. When the microinstruction is executed, every field is decoded and generates control signals. Thus, with N fields, N simultaneous actions are specified. Each action results in the activation of one or more control signals. Generally, but not always, we will want to design the format so that each control signal is activated by no more than one field. Clearly, however, it must be possible for each control signal to be activated by at least one field.

Now consider the individual field. A field consisting of L bits can contain one of 2^L codes, each of which can be encoded to a different control signal pattern. Because only one code can appear in a field at a time, the codes are mutually exclusive, and, therefore, the actions they cause are mutually exclusive.



(a) Direct encoding



(b) Indirect encoding

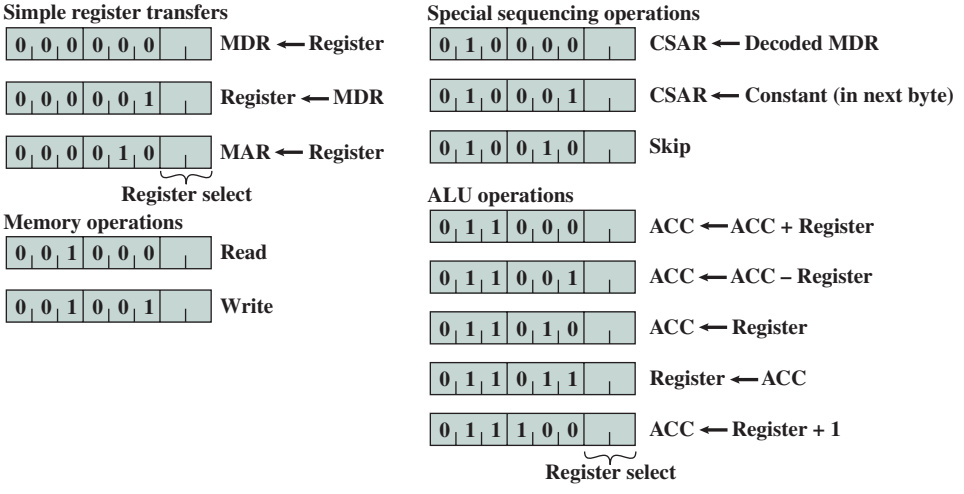
Figure 21.11 Microinstruction Encoding

The design of an encoded microinstruction format can now be stated in simple terms:

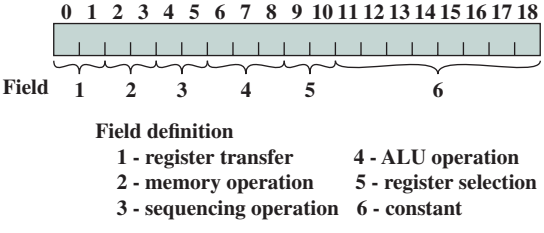
- Organize the format into independent fields. That is, each field depicts a set of actions (pattern of control signals) such that actions from different fields can occur simultaneously.
- Define each field such that the alternative actions that can be specified by the field are mutually exclusive. That is, only one of the actions specified for a given field could occur at a time.

Two approaches can be taken for organizing the encoded microinstruction into fields: functional and resource. The *functional encoding* method identifies functions within the machine and designates fields by function type. For example, if various sources can be used for transferring data to the accumulator, one field can be designated for this purpose, with each code specifying a different source. *Resource encoding* views the machine as consisting of a set of independent resources and devotes one field to each (e.g., I/O, memory, ALU).

Another aspect of encoding is whether it is direct or indirect (Figure 21.11b). With indirect encoding, one field is used to determine the interpretation of another


Simple register transfers
0 0 0 0 0 0
MDR ← Register
0 0 0 0 0 1
Register ← MDR
0 0 0 0 1 0
MAR ← Register
Register select
Memory operations
0 0 1 0 0 0
Read
0 0 1 0 0 1
Write
Special sequencing operations
0 1 0 0 0 0
CSAR ← Decoded MDR
0 1 0 0 0 1
CSAR ← Constant (in next byte)
0 1 0 0 1 0
Skip
ALU operations
0 1 1 0 0 0
ACC ← ACC + Register
0 1 1 0 0 1
ACC ← ACC – Register
0 1 1 0 1 0
ACC ← Register
0 1 1 0 1 1
Register ← ACC
0 1 1 1 0 0
ACC ← Register + 1
Register select

(a) Vertical microinstruction format


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Field 1 2 3 4 5 6
Field definition
1 - register transfer
2 - memory operation
3 - sequencing operation
4 - ALU operation
5 - register selection
6 - constant

(b) Horizontal microinstruction format

Figure 21.12 Alternative Microinstruction Formats for a Simple Machine

field. For example, consider an ALU that is capable of performing eight different arithmetic operations and eight different shift operations. A 1-bit field could be used to indicate whether a shift or arithmetic operation is to be used; a 3-bit field would indicate the operation. This technique generally implies two levels of decoding, increasing propagation delays.

Figure 21.12 is a simple example of these concepts. Assume a processor with a single accumulator and several internal registers, such as a program counter and a temporary register for ALU input. Figure 21.12a shows a highly vertical format. The first 3 bits indicate the type of operation, the next 3 encode the operation, and the final 2 select an internal register. Figure 21.12b is a more horizontal approach, although encoding is still used. In this case, different functions appear in different fields.

LSI-11 Microinstruction Execution

The LSI-11 [SEBE76] is a good example of a vertical microinstruction approach. We look first at the organization of the control unit, then at the microinstruction format.

LSI-11 CONTROL UNIT ORGANIZATION The LSI-11 is the first member of the PDP-11 family that was offered as a single-board processor. The board contains three LSI chips, an internal bus known as the *microinstruction bus* (MIB), and some additional interfacing logic.

Figure 21.13 depicts, in simplified form, the organization of the LSI-11 processor. The three chips are the data, control, and control store chips. The data chip contains an 8-bit ALU, twenty-six 8-bit registers, and storage for several condition

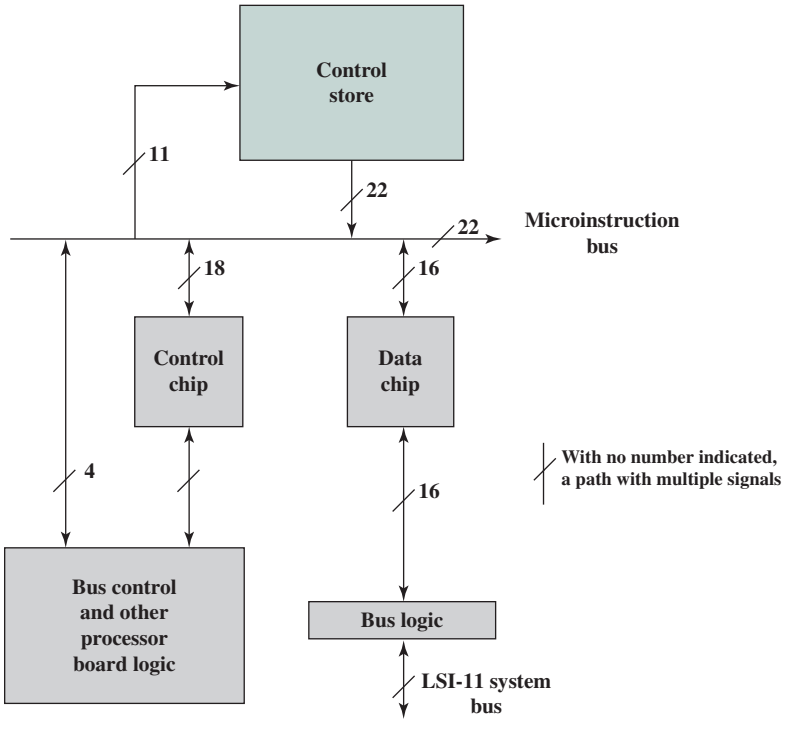


Figure 21.13 Simplified Block Diagram of the LSI-11 Processor

codes. Sixteen of the registers are used to implement the eight 16-bit general-purpose registers of the PDP-11. Others include a program status word, memory address register (MAR), and memory buffer register. Because the ALU deals with only 8 bits at a time, two passes through the ALU are required to implement a 16-bit PDP-11 arithmetic operation. This is controlled by the microprogram.

The control store chip or chips contain the 22-bit-wide control memory. The control chip contains the logic for sequencing and executing microinstructions. It contains the control address register, the control data register, and a copy of the machine instruction register.

The MIB ties all the components together. During microinstruction fetch, the control chip generates an 11-bit address onto the MIB. Control store is accessed, producing a 22-bit microinstruction, which is placed on the MIB. The low-order 16 bits go to the data chip, while the low-order 18 bits go to the control chip. The high-order 4 bits control special processor board functions.

Figure 21.14 provides a still simplified but more detailed look at the LSI-11 control unit: the figure ignores individual chip boundaries. The address sequencing scheme described in Section 21.2 is implemented in two modules. Overall sequence control is provided by the microprogram sequence control module, which is capable

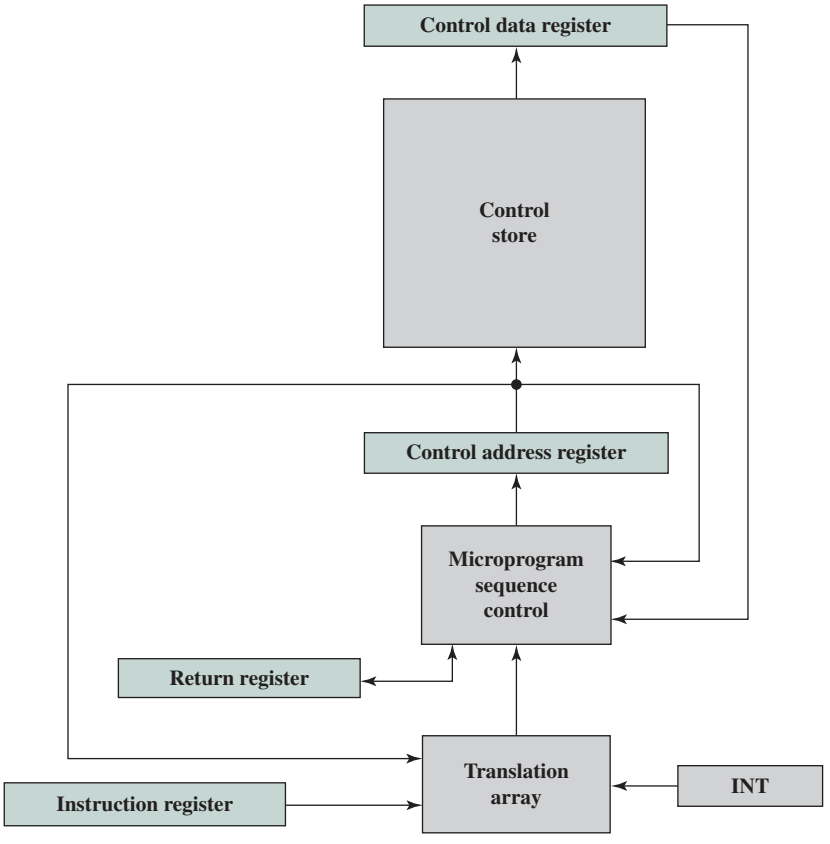


Figure 21.14 Organization of the LSI-11 Control Unit

of incrementing the microinstruction address register and performing unconditional branches. The other forms of address calculation are carried out by a separate translation array. This is a combinatorial circuit that generates an address based on the microinstruction, the machine instruction, the microinstruction program counter, and an interrupt register.

The translation array comes into play on the following occasions:

- The opcode is used to determine the start of a microroutine.
- At appropriate times, address mode bits of the microinstruction are tested to perform appropriate addressing.
- Interrupt conditions are periodically tested.
- Conditional branch microinstructions are evaluated.

LSI-11 MICROINSTRUCTION FORMAT The LSI-11 uses an extremely vertical microinstruction format, which is only 22 bits wide. The microinstruction set strongly resembles the PDP-11 machine instruction set that it implements. This design was intended to optimize the performance of the control unit within the constraint of a vertical, easily programmed design. Table 21.5 lists some of the LSI-11 microinstructions.

Figure 21.15 shows the 22-bit LSI-11 microinstruction format. The high-order 4 bits control special functions on the processor board. The translate bit enables the translation array to check for pending interrupts. The load return register bit is used at the end of a microroutine to cause the next microinstruction address to be loaded from the return register.

The remaining 16 bits are used for highly encoded micro-operations. The format is much like a machine instruction, with a variable-length opcode and one or more operands.

Table 21.5 Some LSI-11 Microinstructions

Arithmetic Operations	General Operations
Add word (byte, literal)	MOV word (byte)
Test word (byte, literal)	Jump
Increment word (byte) by 1	Return
Increment word (byte) by 2	Conditional jump
Negate word (byte)	Set (reset) flags
Conditionally increment (decrement) byte	Load G low
Conditionally add word (byte)	Conditionally MOV word (byte)
Add word (byte) with carry	
Conditionally add digits	**Input/Output Operations**
Subtract word (byte)	Input word (byte)
Compare word (byte, literal)	Input status word (byte)
Subtract word (byte) with carry	Read
Decrement word (byte) by 1	Write
	Read (write) and increment word (byte) by 1
Logical Operations	Read (write) and increment word (byte) by 2
AND word (byte, literal)	Read (write) acknowledge
Test word (byte)	Output word (byte, status)
OR word (byte)	
Exclusive-OR word (byte)	
Bit clear word (byte)	
Shift word (byte) right (left) with (without) carry	
Complement word (byte)	

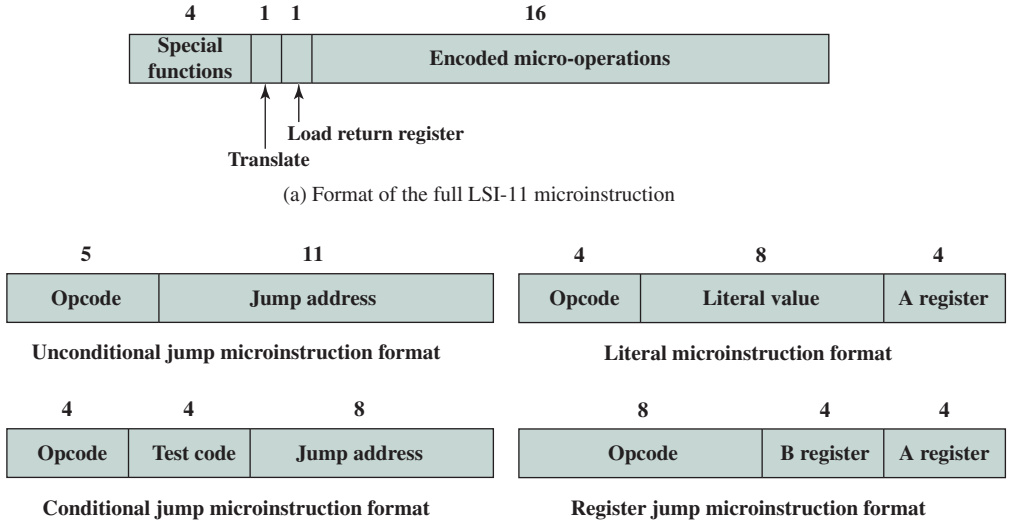


Figure 21.15 LSI-11 Microinstruction Format

IBM 3033 Microinstruction Execution

The standard IBM 3033 control memory consists of 4K words. The first half of these (0000–07FF) contain 108-bit microinstructions, while the remainder (0800–0FFF) are used to store 126-bit microinstructions. The format is depicted in Figure 21.16.

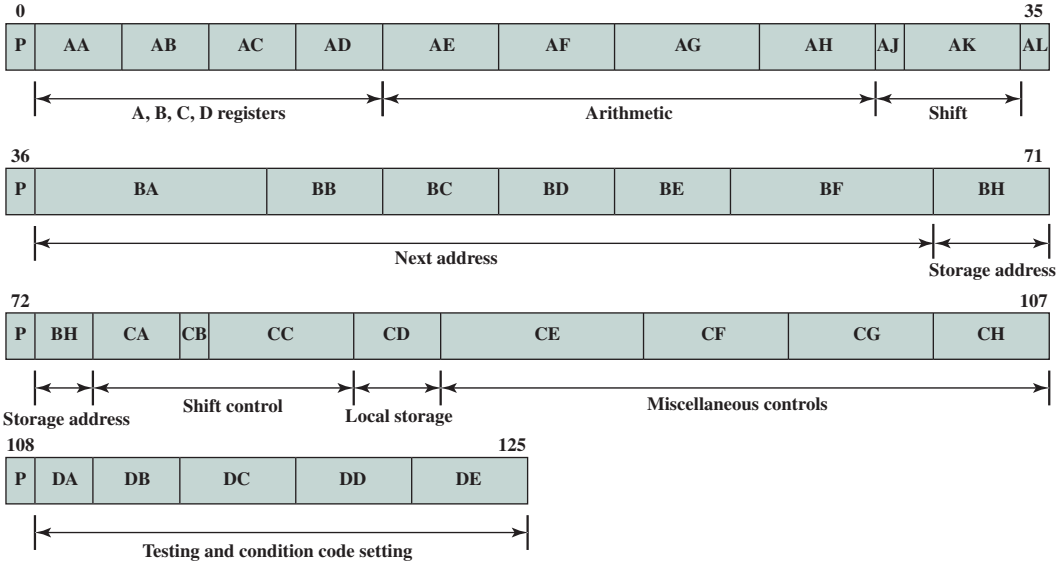


Figure 21.16 IBM 3033 Microinstruction Format

Table 21.6 IBM 3033 Microinstruction Control Fields

ALU Control Fields	
AA(3)	Load A register from one of data registers
AB(3)	Load B register from one of data registers
AC(3)	Load C register from one of data registers
AD(3)	Load D register from one of data registers
AE(4)	Route specified A bits to ALU
AF(4)	Route specified B bits to ALU
AG(5)	Specifies ALU arithmetic operation on A input
AH(4)	Specifies ALU arithmetic operation on B input
AJ(1)	Specifies D or B input to ALU on B side
AK(4)	Route arithmetic output to shifter
CA(3)	Load F register
CB(1)	Activate shifter
CC(5)	Specifies logical and carry functions
CE(7)	Specifies shift amount
Sequencing and Branching Fields	
AL(1)	End operation and perform branch
BA(8)	Set high-order bits (00–07) of control address register
BB(4)	Specifies condition for setting bit 8 of control address register
BC(4)	Specifies condition for setting bit 9 of control address register
BD(4)	Specifies condition for setting bit 10 of control address register
BE(4)	Specifies condition for setting bit 11 of control address register
BF(7)	Specifies condition for setting bit 12 of control address register

Although this is a rather horizontal format, encoding is still extensively used. The key fields of that format are summarized in Table 21.6.

The ALU operates on inputs from four dedicated, non-user-visible registers, A, B, C, and D. The microinstruction format contains fields for loading these registers from user-visible registers, performing an ALU function, and specifying a user-visible register for storing the result. There are also fields for loading and storing data between registers and memory.

The sequencing mechanism for the IBM 3033 was discussed in Section 21.2.

21.4 TI 8800

The Texas Instruments 8800 Software Development Board (SDB) is a microprogrammable 32-bit computer card. The system has a writable control store, implemented in RAM rather than ROM. Such a system does not achieve the speed or

density of a microprogrammed system with a ROM control store. However, it is useful for developing prototypes and for educational purposes.

The 8800 SDB consists of the following components (Figure 21.17):

- Microcode memory
- Microsequencer
- 32-bit ALU
- Floating-point and integer processor
- Local data memory

Two buses link the internal components of the system. The DA bus provides data from the microinstruction data field to the ALU, the floating-point processor, or the microsequencer. In the latter case, the data consists of an address to be used for a branch instruction. The bus can also be used for the ALU or microsequencer to

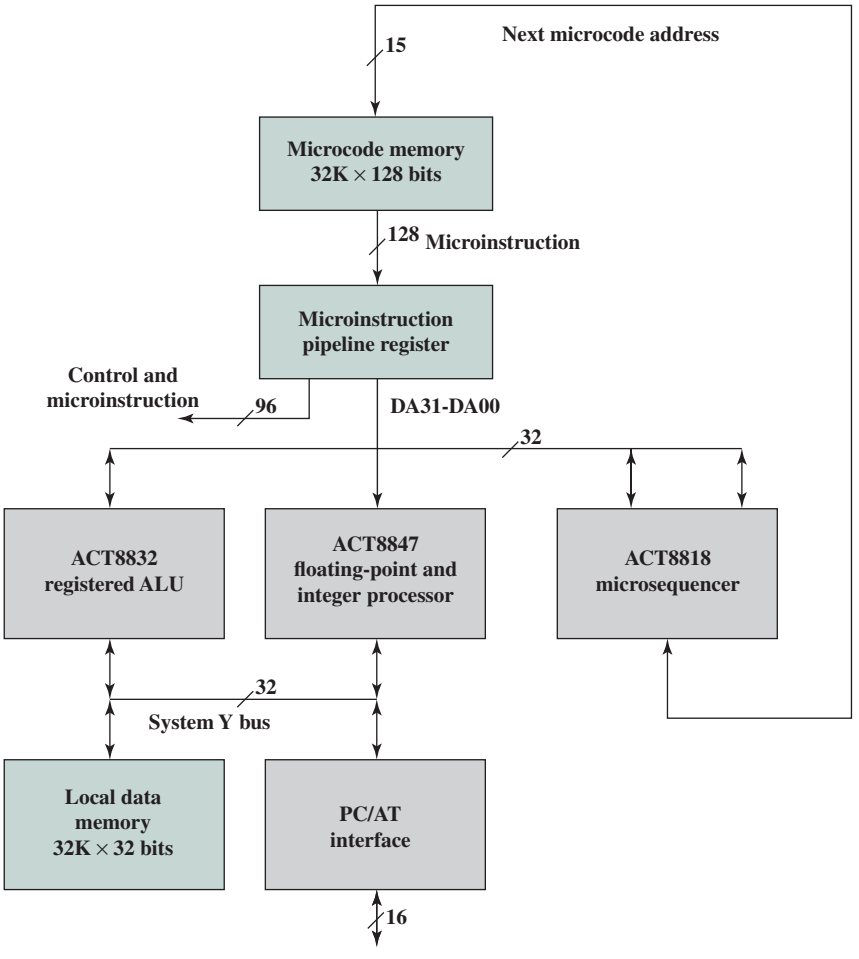


Figure 21.17 TI 8800 Block Diagram

provide data to other components. The system Y bus connects the ALU and floating-point processor to local memory and to external modules via the PC interface.

The board fits into an IBM PC-compatible host computer. The host computer provides a suitable platform for microcode assembly and debug.

Microinstruction Format

The microinstruction format for the 8800 consists of 128 bits broken down into 30 functional fields, as indicated in Table 21.7. Each field consists of one or more bits, and the fields are grouped into five major categories:

- Control of board
- 8847 floating-point and integer processor chip
- 8832 registered ALU
- 8818 microsequencer
- WCS data field

As indicated in Figure 21.17, the 32 bits of the WCS data field are fed into the DA bus to be provided as data to the ALU, floating-point processor, or microsequencer. The other 96 bits (fields 1–27) of the microinstruction are control signals that are fed directly to the appropriate module. For simplicity, these other connections are not shown in Figure 21.17.

The first six fields deal with operations that pertain to the control of the board, rather than controlling an individual component. Control operations include the following:

- Selecting condition codes for sequencer control. The first bit of field 1 indicates whether the condition flag is to be set to 1 or 0, and the remaining 4 bits indicate which flag is to be set.
- Sending an I/O request to the PC/AT.
- Enabling local data memory read/write operations.
- Determining the unit driving the system Y bus. One of the four devices attached to the bus (Figure 21.17) is selected.

The last 32 bits are the data field, which contain information specific to a particular microinstruction.

The remaining fields of the microinstruction are best discussed in the context of the device that they control. In the remainder of this section, we discuss the microsequencer and the registered ALU. The floating-point unit introduces no new concepts and is skipped.

Microsequencer

The principal function of the 8818 microsequencer is to generate the next microinstruction address for the microprogram. This 15-bit address is provided to the microcode memory (Figure 21.17).

The next address can be selected from one of five sources:

1. The microprogram counter (MPC) register, used for repeat (reuse same address) and continue (increment address by 1) instructions.

Table 21.7 TI 8800 Microinstruction Format

Field Number	Number of Bits	Description
		Control of Board
1	5	Select condition code input
2	1	Enable/disable external I/O request signal
3	2	Enable/disable local data memory read/write operations
4	1	Load status/do no load status
5	2	Determine unit driving Y bus
6	2	Determine unit driving DA bus
		8847 Floating-Point and Integer Processing Chip
7	1	C register control: clock, do not clock
8	1	Select most significant or least significant bits for Y bus
9	1	C register data source: ALU, multiplexer
10	4	Select IEEE or FAST mode for ALU and MUL
11	8	Select sources for data operands: RA registers, RB registers, P register, 5 register, C register
12	1	RB register control: clock, do not clock
13	1	RA register control: clock, do not clock
14	2	Data source confirmation
15	2	Enable/disable pipeline registers
16	11	8847 ALU function
		8832 Registered ALU
17	2	Write enable/disable data output to selected register: most significant half, least significant half
18	2	Select register file data source: DA bus, DB bus, ALU Y MUX output, system Y bus
19	3	Shift instruction modifier
20	1	Carry in: force, do not force
21	2	Set ALU configuration mode: 32, 16, or 8 bits
22	2	Select input to 5 multiplexer: register file, DB bus, MQ register
23	1	Select input to R multiplexer: register file, DA bus
24	6	Select register in file C for write
25	6	Select register in file B for read
26	6	Select register in file A for write
27	8	ALU function
		8818 Microsequencer
28	12	Control input signals to the 8818
		WCS Data Field
29	16	Most significant bits of writable control store data field
30	16	Least significant bits of writable control store data field

2. The stack, which supports microprogram subroutine calls as well as iterative loops and returns from interrupts.
3. The DRA and DRB ports, which provide two additional paths from external hardware by which microprogram addresses can be generated. These two ports are connected to the most significant and least significant 16 bits of the DA bus, respectively. This allows the microsequencer to obtain the next instruction address from the WCS data field of the current microinstruction or from a result calculated by the ALU.
4. Register counters RCA and RCB, which can be used for additional address storage.
5. An external input onto the bidirectional Y port to support external interrupts.

Figure 21.18 is a logical block diagram of the 8818. The device consists of the following principal functional groups:

- A 16-bit microprogram counter (MPC) consisting of a register and an incrementer.
- Two register counters, RCA and RCB, for counting loops and iterations, storing branch addresses, or driving external devices.
- A 65-word by 16-bit stack, which allows microprogram subroutine calls and interrupts.
- An interrupt return register and Y output enable for interrupt processing at the microinstruction level.
- A Y output multiplexer by which the next address can be selected from MPC, RCA, RCB, external buses DRA and DRB, or the stack.

REGISTERS/COUNTERS The registers RCA and RCB may be loaded from the DA bus, either from the current microinstruction or from the output of the ALU. The values may be used as counters to control the flow of execution and may be automatically decremented when accessed. The values may also be used as microinstruction addresses to be supplied to the Y output multiplexer. Independent control of both registers during a single microinstruction cycle is supported with the exception of simultaneous decrement of both registers.

STACK The stack allows multiple levels of nested calls or interrupts, and it can be used to support branching and looping. Keep in mind that these operations refer to the control unit, not the overall processor, and that the addresses involved are those of microinstructions in the control memory.

Six stack operations are possible:

1. Clear, which sets the stack pointer to zero, emptying the stack;
2. Pop, which decrements the stack pointer;
3. Push, which puts the contents of the MPC, interrupt return register, or DRA bus onto the stack and increments the stack pointer;
4. Read, which makes the address indicated by the read pointer available at the Y output multiplexer;

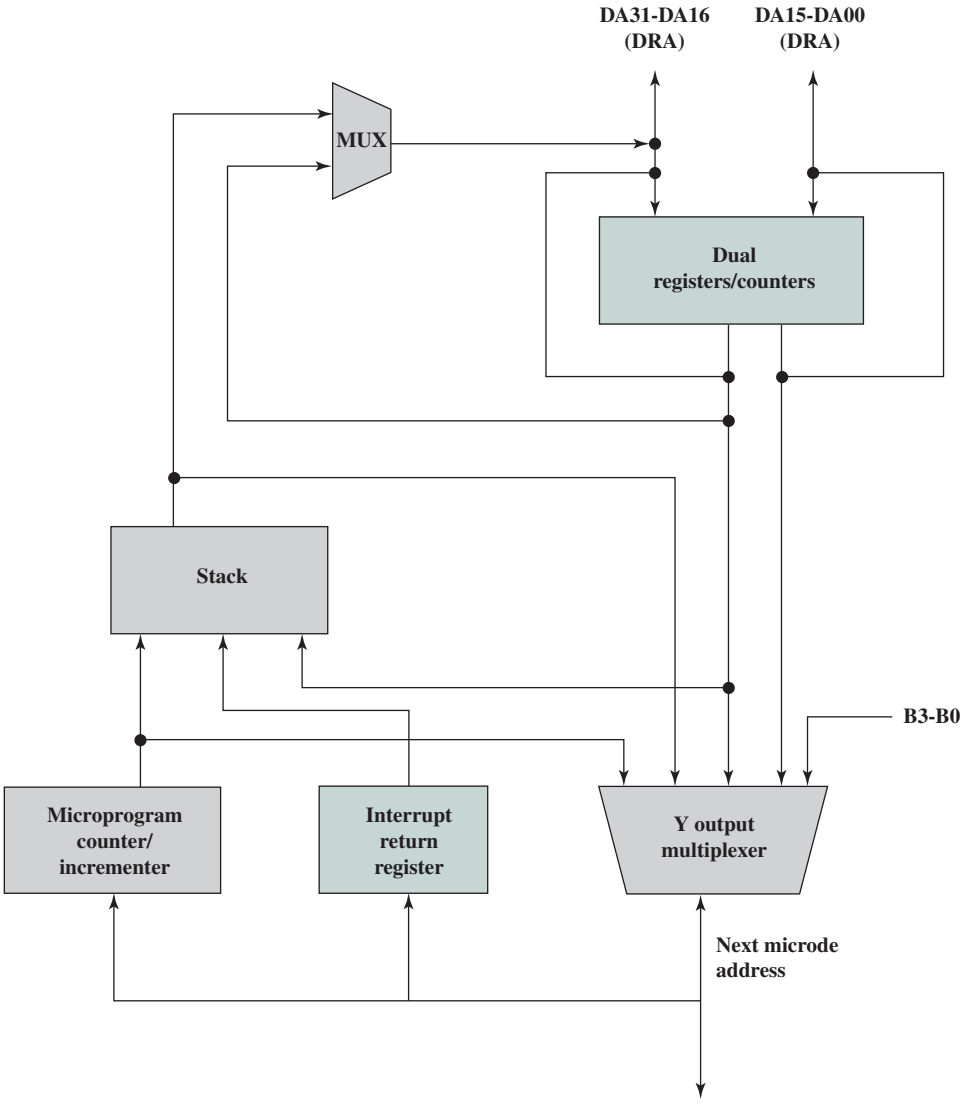


Figure 21.18 TI 8818 Microsequencer

5. Hold, which causes the address of the stack pointer to remain unchanged;
6. Load stack pointer, which inputs the seven least significant bits of DRA to the stack pointer.

CONTROL OF MICROSEQUENCER The microsequencer is controlled primarily by the 12-bit field of the current microinstruction, field 28 (Table 21.7). This field consists of the following subfields:

- **OSEL (1 bit):** Output select. Determines which value will be placed on the output of the multiplexer that feeds into the DRA bus (upper-left-hand corner of

Figure 21.18). The output is selected to come from either the stack or from register RCA. DRA then serves as input to either the Y output multiplexer or to register RCA.

- **SELDR (1 bit):** Select DR bus. If set to 1, this bit selects the external DA bus as input to the DRA/DRB buses. If set to 0, selects the output of the DRA multiplexer to the DRA bus (controlled by OSEL) and the contents of RCB to the DRB bus.
- **ZEROIN (1 bit):** Used to indicate a conditional branch. The behavior of the microsequencer will then depend on the condition code selected in field 1 (Table 21.7).
- **RC2–RC0 (3 bits):** Register controls. These bits determine the change in the contents of registers RCA and RCB. Each register can either remain the same, decrement, or load from the DRA/DRB buses.
- **S2–S0 (3 bits):** Stack controls. These bits determine which stack operation is to be performed.
- **MUX2–MUX0:** Output controls. These bits, together with the condition code if used, control the Y output multiplexer and therefore the next microinstruction address. The multiplexer can select its output from the stack, DRA, DRB, or MPC.

These bits can be set individually by the programmer. However, this is typically not done. Rather, the programmer uses mnemonics that equate to the bit patterns that would normally be required. Table 21.8 lists the 15 mnemonics for field 28. A microcode assembler converts these into the appropriate bit patterns.

Table 21.8 TI 8818 Microsequencer Microinstruction Bits (Field 28)

Mnemonic	Value	Description
RST8818	000000000110	Reset Instruction
BRA88181	011000111000	Branch to DRA Instruction
BRA88180	010000111110	Branch to DRA Instruction
INC88181	000000111110	Continue Instruction
INC88180	001000001000	Continue Instruction
CAL88181	010000110000	Jump to Subroutine at Address Specified by DRA
CAL88180	010000101110	Jump to Subroutine at Address Specified by DRA
RET8818	000000011010	Return from Subroutine
PUSH8818	000000110111	Push Interrupt Return Address onto Stack
POP8818	100000010000	Return from Interrupt
LOADDRA	000010111110	Load DRA Counter from DA Bus
LOADDRB	000110111110	Load DRB Counter from DA Bus
LOADDRAB	000110111100	Load DRA/DRB
DECRDRA	010001111100	Decrement DRA Counter and Branch If Not Zero
DECRDRB	010101111100	Decrement DRB Counter and Branch If Not Zero

As an example, the instruction INC88181 is used to cause the next microinstruction in sequence to be selected, if the currently selected condition code is 1. From Table 21.8, we have

$$\text{INC88181} = 000000111110$$

which decodes directly into

- **OSEL = 0:** Selects RCA as output from DRA output MUX; in this case the selection is irrelevant.
- **SELDR = 0:** As defined previously; again, this is irrelevant for this instruction.
- **ZEROIN = 0:** Combined with the value for MUX, indicates no branch should be taken.
- **R = 000:** Retain current value of RA and RC.
- **S = 111:** Retain current state of stack.
- **MUX = 110:** Choose MPC when condition code = 1, DRA when condition code = 0.

Registered ALU

The 8832 is a 32-bit ALU with 64 registers that can be configured to operate as four 8-bit ALUs, two 16-bit ALUs, or a single 32-bit ALU.

The 8832 is controlled by the 39 bits that make up fields 17 through 27 of the microinstruction (Table 21.7); these are supplied to the ALU as control signals. In addition, as indicated in Figure 21.17, the 8832 has external connections to the 32-bit DA bus and the 32-bit system Y bus. Inputs from the DA can be provided simultaneously as input data to the 64-word register file and to the ALU logic module. Input from the system Y bus is provided to the ALU logic module. Results of the ALU and shift operations are output to the DA bus or the system Y bus. Results can also be fed back to the internal register file.

Three 6-bit address ports allow a two-operand fetch and an operand write to be performed within the register file simultaneously. An MQ shifter and MQ register can also be configured to function independently to implement double-precision 8-bit, 16-bit, and 32-bit shift operations.

Fields 17 through 26 of each microinstruction control the way in which data flows within the 8832 and between the 8832 and the external environment. The fields are as follows:

17. **Write Enable.** These two bits specify write 32 bits, 16 most significant bits, 16 least significant bits, or do not write into register file. The destination register is defined by field 24.
18. **Select Register File Data Source.** If a write is to occur to the register file, these two bits specify the source: DA bus, DB bus, ALU output, or system Y bus.
19. **Shift Instruction Modifier.** Specifies options concerning supplying end fill bits and reading bits that are shifted during shift instructions.
20. **Carry In.** This bit indicates whether a bit is carried into the ALU for this operation.

21. **ALU Configuration Mode.** The 8832 can be configured to operate as a single 32-bit ALU, two 16-bit ALUs, or four 8-bit ALUs.
22. **S Input.** The ALU logic module inputs are provided by two internal multiplexers referred to as the S and R multiplexers. This field selects the input to be provided by the S multiplexer: register file, DB bus, or MQ register. The source register is defined by field 25.
23. **R Input.** Selects input to be provided by the R multiplexer: register file or DA bus.
24. **Destination Register.** Address of register in register file to be used for the destination operand.
25. **Source Register.** Address of register in register file to be used for the source operand, provided by the S multiplexer.
26. **Source Register.** Address of register in register file to be used for the source operand, provided by the R multiplexer.

Finally, field 27 is an 8-bit opcode that specifies the arithmetic or logical function to be performed by the ALU. Table 21.9 lists the different operations that can be performed.

As an example of the coding used to specify fields 17 through 27, consider the instruction to add the contents of register 1 to register 2 and place the result in register 3. The symbolic instruction is

```
CONT11 [17], WELH, SELRYFYMX, [24], R3, R2, R1, PASS + ADD
```

The assembler will translate this into the appropriate bit pattern. The individual components of the instruction can be described as follows:

- CONT11 is the basic NOP instruction.
- Field [17] is changed to WELH (write enable, low and high), so that a 32-bit register is written into.
- Field [18] is changed to SELRFYMX to select the feedback from the ALU Y MUX output.
- Field [24] is changed to designate register R3 for the destination register.
- Field [25] is changed to designate register R2 for one of the source registers.
- Field [26] is changed to designate register R1 for one of the source registers.
- Field [27] is changed to specify an ALU operation of ADD. The ALU shifter instruction is PASS; therefore, the ALU output is not shifted by the shifter.

Several points can be made about the symbolic notation. It is not necessary to specify the field number for consecutive fields. That is,

```
CONT11 [17],WELH,[18],SELRFYMX
```

can be written as

```
CONT11 [17],WELH,SELRFYMX
```

because SELRFYMX is in field 18.

ALU instructions from Group 1 of Table 21.9 must always be used in conjunction with Group 2. ALU instructions from Groups 3 to 5 must not be used with Group 2.

Table 21.9 TI 8832 Registered ALU Instruction Field (Field 27)

Group 1		**Function**
ADD	H#01	R + S + Cn
SUBR	H#02	(NOT R) + S + Cn
SUBS	H#03	R = (NOT S) + Cn
INSC	H#04	S + Cn
INCNS	H#05	(NOT S) + Cn
INCR	H#06	R + Cn
INCNR	H#07	(NOT R) + Cn
XOR	H#09	R XOR S
AND	H#0A	R AND S
OR	H#0B	R OR S
NAND	H#0C	R NAND S
NOR	H#0D	R NOR S
ANDNR	H#0E	(NOT R) AND S
Group 2		**Function**
SRA	H#00	Arithmetic right single precision shift
SRAD	H#10	Arithmetic right double precision shift
SRL	H#20	Logical right single precision shift
SRLD	H#30	Logical right double precision shift
SLA	H#40	Arithmetic left single precision shift
SLAD	H#50	Arithmetic left double precision shift
SLC	H#60	Circular left single precision shift
SLCD	H#70	Circular left double precision shift
SRC	H#80	Circular right single precision shift
SRCD	H#90	Circular right double precision shift
MQSRA	H#A0	Arithmetic right shift MQ register
MQSRL	H#B0	Logical right shift MQ register
MQSLL	H#C0	Logical left shift MQ register
MQSLC	H#D0	Circular left shift MQ register
LOADMQ	H#E0	Load MQ register
PASS	H#F0	Pass ALU to Y (no shift operation)
Group 3		**Function**
SET1	H#08	Set bit 1
Set0	H#18	Set bit 0
TB1	H#28	Test bit 1
TB0	H#38	Test bit 0
ABS	H#48	Absolute value
SMTC	H#58	Sign magnitude/twos-complement

Group 3		Function
ADDI	H#68	Add immediate
SUBI	H#78	Subtract immediate
BADD	H#88	Byte add R to S
BSUBS	H#98	Byte subtract S from R
BSUBR	H#A8	Byte subtract R from S
BINCS	H#B8	Byte increment S
BINCNS	H#C8	Byte increment negative S
BXOR	H#D8	Byte XOR R and S
BAND	H#E8	Byte AND R and S
BOR	H#F8	Byte OR R and S
Group 4		**Function**
CRC	H#00	Cyclic redundancy character accum.
SEL	H#10	Select S or R
SNORM	H#20	Single length normalize
DNORM	H#30	Double length normalize
DIVRF	H#40	Divide remainder fix
SDIVQF	H#50	Signed divide quotient fix
SMULI	H#60	Signed multiply iterate
SMULT	H#70	Signed multiply terminate
SDIVIN	H#80	Signed divide initialize
SDIVIS	H#90	Signed divide start
SDIVI	H#A0	Signed divide iterate
UDIVIS	H#B0	Unsigned divide start
UDIVI	H#C0	Unsigned divide iterate
UMULI	H#D0	Unsigned multiply iterate
SDIVIT	H#E0	Signed divide terminate
UDIVIT	H#F0	Unsigned divide terminate
Group 5		**Function**
LOADFF	H#0F	Load divide/BCD flip-flops
CLR	H#1F	Clear
DUMPFF	H#5F	Output divide/BCD flip-flops
BCDBIN	H#7F	BCD to binary
EX3BC	H#8F	Excess (3 byte correction
EX3C	H#9F	Excess (3 word correction
SDIVO	H#AF	Signed divide overflow test
BINEX3	H#DF	Binary to excess − 3
NOP32	H#FF	No operation

21.5 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

control memory	microinstruction encoding	microprogrammed control unit
control word	microinstruction execution	microprogramming language
firmware	microinstruction sequencing	soft microprogramming
hard microprogramming	microinstructions	unpacked microinstruction
horizontal microinstruction	microprogram	vertical microinstruction

Review Questions

21.1 What is the difference between a hardwired implementation and a microprogrammed implementation of a control unit?
21.2 How is a horizontal microinstruction interpreted?
21.3 What is the purpose of a control memory?
21.4 What is a typical sequence in the execution of a horizontal microinstruction?
21.5 What is the difference between horizontal and vertical microinstructions?
21.6 What are the basic tasks performed by a microprogrammed control unit?
21.7 What is the difference between packed and unpacked microinstructions?
21.8 What is the difference between hard and soft microprogramming?
21.9 What is the difference between functional and resource encoding?
21.10 List some common applications of microprogramming.

Problems

21.1 Describe the implementation of the multiply instruction in the hypothetical machine designed by Wilkes. Use narrative and a flowchart.

21.2 Assume a microinstruction set that includes a microinstruction with the following symbolic form:

$$\text{IF } (AC_0 = 1) \text{ THEN CAR} \leftarrow (C_{0-6}) \text{ ELSE CAR} \leftarrow (\text{CAR}) + 1$$

where AC_0 is the sign bit of the accumulator and C_{0-6} are the first seven bits of the microinstruction. Using this microinstruction, write a microprogram that implements a Branch Register Minus (BRM) machine instruction, which branches if the AC is negative. Assume that bits C_1 through C_n of the microinstruction specify a parallel set of micro-operations. Express the program symbolically.

21.3 A simple processor has four major phases to its instruction cycle: fetch, indirect, execute, and interrupt. Two 1-bit flags designate the current phase in a hardwired implementation.
a. Why are these flags needed?
b. Why are they not needed in a microprogrammed control unit?

21.4 Consider the control unit of Figure 21.7. Assume that the control memory is 24 bits wide. The control portion of the microinstruction format is divided into two fields. A micro-operation field of 13 bits specifies the micro-operations to be performed. An address selection field specifies a condition, based on the flags, that will cause a microinstruction branch. There are eight flags.

a. How many bits are in the address selection field?
b. How many bits are in the address field?
c. What is the size of the control memory?

21.5 How can unconditional branching be done under the circumstances of the previous problem? How can branching be avoided; that is, describe a microinstruction that does not specify any branch, conditional or unconditional.

21.6 We wish to provide 8 control words for each machine instruction routine. Machine instruction opcodes have 5 bits, and control memory has 1024 words. Suggest a mapping from the instruction register to the control address register.

21.7 An encoded microinstruction format is to be used. Show how a 9-bit micro-operation field can be divided into subfields to specify 46 different actions.

21.8 A processor has 16 registers, an ALU with 16 logic and 16 arithmetic functions, and a shifter with 8 operations, all connected by an internal processor bus. Design a microinstruction format to specify the various micro-operations for the processor.

APPENDIX A

PROJECTS FOR TEACHING COMPUTER ORGANIZATION AND ARCHITECTURE

A.1 Interactive Simulations

A.2 Research Projects

A.3 Simulation Projects
SimpleScalar
SMPCache

A.4 Assembly Language Projects

A.5 Reading/Report Assignments

A.6 Writing Assignments

A.7 Test Bank

Many instructors believe that research or implementation projects are crucial to the clear understanding of the concepts of computer organization and architecture. Without projects, it may be difficult for students to grasp some of the basic concepts and interactions among components. Projects reinforce the concepts introduced in the book, give students a greater appreciation of the inner workings of processors and computer systems, and can motivate students and give them confidence that they have mastered the material.

In this text, I have tried to present the concepts of computer organization and architecture as clearly as possible and have provided numerous homework problems to reinforce those concepts. Many instructors will wish to supplement this material with projects. This appendix provides some guidance in that regard and describes support material available in the **Instructor's Resource Center (IRC)** for this book, accessible by instructors online from Pearson. The support material covers six types of projects and other student exercises:

- Interactive simulations
- Research projects
- Simulation projects
- Assembly language projects
- Reading/report assignments
- Writing assignments
- Test bank

A.1 INTERACTIVE SIMULATIONS

Interactive simulations provide a powerful tool for understanding the complex design features of a modern computer system. Today's students want to be able to visualize the various complex computer systems mechanisms on their own computer screen. A total of 20 simulations are used to illustrate key functions and algorithms in computer organization and architecture design. Table A.1 lists the simulations by chapter. At the relevant point in the book, an icon indicates that a relevant interactive simulation is available online for student use.

Because the simulations enable the user to set initial conditions, they can serve as the basis for student assignments. The IRC for this book includes a set of assignments, one set for each of the interactive simulations. Each assignment includes several specific problems that can be assigned to students.

The interactive simulations were developed under the direction of Professor Israel Koren, at the University of Massachusetts Department of Electrical and Computer Engineering. Aswin Sreedhar of the University of Massachusetts developed the interactive simulation assignments. For access to the animations, click on the rotating globe at this book's web site at http://williamstallings.com/ComputerOrganization.

Table A.1 Computer Organization and Architecture—Interactive Simulations by Chapter

Chapter 4—Cache Memory	
Cache Simulator	Emulates small-sized caches based on a user-input cache model and displays the cache contents at the end of the simulation cycle based on an input sequence which is entered by the user, or randomly generated if so selected.
Cache Time Analysis	Demonstrates Average Memory Access Time analysis for the cache parameters you specify.
Multitask Cache Demonstrator	Models cache on a system that supports multitasking.
Selective Victim Cache Simulator	Compares three different cache policies.
Chapter 5—Internal Memory	
Interleaved Memory Simulator	Demonstrates the effect of interleaving memory.
Chapter 6—External Memory	
RAID	Determine storage efficiency and reliability.
Chapter 7—Input/Output	
I/O System Design Tool	Evaluates comparative cost and performance of different I/O systems.
Chapter 8—OS Support	
Page Replacement Algorithms	Compares LRU, FIFO, and Optimal.
More Page Replacement Algorithms	Compares a number of policies.
Chapter 14—CPU Structure and Function	
Reservation Table Analyzer	Evaluates reservation tables. which are a way of representing the task flow pattern of a pipelined system.
Branch Prediction	Demonstrates three different branch prediction schemes.
Branch Target Buffer	Combined branch predictor/branch target buffer simulator.
Chapter 15—Reduced Instruction Set Computers	
MIPS 5-Stage Pipeline	Simulates the pipeline.
Loop Unrolling	Simulates the loop unrolling software technique for exploiting instruction-level parallelism.
Chapter 16—Instruction-Level Parallelism and Superscalar Processors	
Pipeline with Static vs. Dynamic Scheduling	A more complex simulation of the MIPS pipeline.
Reorder Buffer Simulator	Simulates instruction reordering in a RISC pipeline.
Scoreboarding Technique for Dynamic Scheduling	Simulation of an instruction scheduling technique used in a number of processors.
Tomasulo's Algorithm	Simulation of another instruction scheduling technique.
Alternative Simulation of Tomasulo's Algorithm	Another simulation of Tomasulo's algorithm.
Chapter 17—Parallel Processing	
Vector Processor Simulation	Demonstrates execution of vector processing instructions.

A.2 RESEARCH PROJECTS

An effective way of reinforcing basic concepts from the course and for teaching students research skills is to assign a research project. Such a project could involve a literature search as well as a Web search of vendor products, research lab activities, and standardization efforts. Projects could be assigned to teams or, for smaller projects, to individuals. In any case, it is best to require some sort of project proposal early in the term, giving the instructor time to evaluate the proposal for appropriate topic and appropriate level of effort. Student handouts for research projects should include

- A format for the proposal
- A format for the final report
- A schedule with intermediate and final deadlines
- A list of possible project topics

The students can select one of the listed topics or devise their own comparable project. The IRC includes a suggested format for the proposal and final report as well as a list of possible research topics.

A.3 SIMULATION PROJECTS

An excellent way to obtain a grasp of the internal operation of a processor and to study and appreciate some of the design trade-offs and performance implications is by simulating key elements of the processor. Two tools that are useful for this purpose are SimpleScalar and SMPCache.

Compared with actual hardware implementation, simulation provides two advantages for both research and educational use:

- With simulation, it is easy to modify various elements of an organization, to vary the performance characteristics of various components, and then to analyze the effects of such modifications.
- Simulation provides for detailed performance statistics collection, which can be used to understand performance trade-offs.

SimpleScalar

SimpleScalar [BURG97, MANJ01a, MANJ01b] is a set of tools that can be used to simulate real programs on a range of modern processors and systems. The tool set includes compiler, assembler, linker, and simulation and visualization tools. SimpleScalar provides processor simulators that range from an extremely fast functional simulator to a detailed out-of-order issue, superscalar processor simulator that supports nonblocking caches and speculative execution. The instruction set architecture and organizational parameters may be modified to create a variety of experiments.

The IRC for this book includes a concise introduction to SimpleScalar for students, with instructions on how to load and get started with SimpleScalar. The manual also includes some suggested project assignments.

SimpleScalar is a portable software package the runs on most UNIX platforms. The SimpleScalar software can be downloaded from the SimpleScalar Web site. It is available at no cost for noncommercial use.

SMPCache

SMPCache is a trace-driven simulator for the analysis and teaching of cache memory systems on symmetric multiprocessors [RODR01]. The simulation is based on a model built according to the architectural basic principles of these systems. The simulator has a full graphic and friendly interface. Some of the parameters that can be studied with the simulator are: program locality; influence of the number of processors, cache coherence protocols, schemes for bus arbitration, mapping, replacement policies, cache size (blocks in cache), number of cache sets (for set associative caches), and number of words by block (memory block size).

The IRC for this book includes a concise introduction to SMPCache for students, with instructions on how to load and get started with SMPCache. The manual also includes some suggested project assignments.

SMPCache is a portable software package the runs on PC systems with Windows. The SMPCache software can be downloaded from the SMPCache Web site. It is available at no cost for noncommercial use.

A.4 ASSEMBLY LANGUAGE PROJECTS

Assembly language programming is often used to teach students low-level hardware components and computer architecture basics. CodeBlue is a simplified assembly language program developed at the U.S. Air Force Academy. The goal of the work was to develop and teach assembly language concepts using a visual simulator that students can learn in a single class. The developers also wanted students to find the language motivational and fun to use. The CodeBlue language is much simpler than most simplified architecture instruction sets such as the SC123. Still it allows students to develop interesting assembly level programs that compete in tournaments, similar to the far more complex SPIMbot simulator. Most important, through CodeBlue programming, students learn fundamental computer architecture concepts such as instructions and data co-residence in memory, control structure implementation, and addressing modes.

To provide a basis for projects, the developers have built a visual development environment that allows students to create a program, see its representation in memory, step through the program's execution, and simulate a battle of competing programs in a visual memory environment.

Projects can be built around the concept of a Core War tournament. Core War is a programming game introduced to the public in the early 1980s, which was popular for a period of 15 years or so. Core War has four main components: a memory array of 8000 addresses, a simplified assembly language Redcode, an executive program called MARS (an acronym for Memory Array Redcode Simulator), and the set of contending battle programs. Two battle programs are entered into the memory array at randomly chosen positions; neither program knows where the

other one is. MARS executes the programs in a simple version of time-sharing. The two programs take turns: a single instruction of the first program is executed, then a single instruction of the second, and so on. What a battle program does during the execution cycles allotted to it is entirely up to the programmer. The aim is to destroy the other program by ruining its instructions. The CodeBlue environment substitutes CodeBlue for Redcode and provides its own interactive execution interface.

The IRC includes the CodeBlue environment, a user's manual for students, other supporting material, and suggested assignments.

A.5 READING/REPORT ASSIGNMENTS

Another excellent way to reinforce concepts from the course and to give students research experience is to assign papers from the literature to be read and analyzed. The IRC includes a suggested list of papers to be assigned, organized by chapter. The Premium Content Web site provides a copy of each of the papers. The IRC also includes a suggested assignment wording.

A.6 WRITING ASSIGNMENTS

Writing assignments can have a powerful multiplier effect in the learning process in a technical discipline such as computer organization and architecture. Adherents of the Writing Across the Curriculum (WAC) movement (**http://wac.colostate.edu/**) report substantial benefits of writing assignments in facilitating learning. Writing assignments lead to more detailed and complete thinking about a particular topic. In addition, writing assignments help to overcome the tendency of students to pursue a subject with a minimum of personal engagement, just learning facts and problem-solving techniques without obtaining a deep understanding of the subject matter.

The IRC contains a number of suggested writing assignments, organized by chapter. Instructors may ultimately find that this is the most important part of their approach to teaching the material. I would greatly appreciate any feedback on this area and any suggestions for additional writing assignments.

A.7 TEST BANK

A test bank for the book is available at the IRC site for this book. For each chapter, the test bank includes true/false, multiple choice, and fill-in-the-blank questions. The test bank is an effective way to assess student comprehension of the material.

APPENDIX B

ASSEMBLY LANGUAGE AND RELATED TOPICS

B.1 Assembly Language
- Assembly Language Elements
- Type of Assembly Language Statements
- Example: Greatest Common Divisor Program

B.2 Assemblers
- Two-Pass Assembler
- One-Pass Assembler
- Example: Prime Number Program

B.3 Loading and Linking
- Relocation
- Loading
- Linking

B.4 Key Terms, Review Questions, and Problems

The topic of assembly language was briefly introduced in Chapter 13. This appendix provides more detail and also covers a number of related topics. There are a number of reasons why it is worthwhile to study assembly language programming (as compared with programming in a higher-level language), including the following:

1. It clarifies the execution of instructions.
2. It shows how data are represented in memory.
3. It shows how a program interacts with the operating system, processor, and the I/O system.
4. It clarifies how a program accesses external devices.
5. Understanding assembly language programmers makes students better high-level language (HLL) programmers, by giving them a better idea of the target language that the HLL must be translated into.

We begin this chapter with a study of the basic elements of an assembly language, using the x86 architecture for our examples.[1] Next, we look at the operation of the assembler. This is followed by a discussion of linkers and loaders.

Table B.1 defines some of the key terms used in this appendix.

B.1 ASSEMBLY LANGUAGE

Assembly language is a programming language that is one step away from machine language. Typically, each assembly language instruction is translated into one machine instruction by the assembler. Assembly language is hardware dependent, with a different assembly language for each type of processor. In particular, assembly language instructions can make reference to specific registers in the processor, include all of the opcodes of the processor, and reflect the bit length of the various registers of the processor and operands of the machine language. An assembly language programmer must therefore understand the computer's architecture.

Programmers rarely use assembly language for applications or even systems programs. HLLs provide an expressive power and conciseness that greatly eases the programmer's tasks. The disadvantages of using an assembly language rather than an HLL include the following [FOG08]:

1. **Development time.** Writing code in assembly language takes much longer than writing in a high-level language.
2. **Reliability and security.** It is easy to make errors in assembly code. The assembler is not checking if the calling conventions and register save conventions are obeyed. Nobody is checking for you if the number of PUSH and POP instructions is the same in all possible branches and paths. There are so many possibilities for hidden errors in assembly code that it affects the reliability and security of the project unless you have a very systematic approach to testing and verifying.

[1]There are a number of assemblers for the x86 architecture. Our examples use NASM (Netwide Assembler), an open source assembler. A copy of the NASM manual is at this book's Premium Content site.

Table B.1 Key Terms for this Appendix

Assembler

A program that translates assembly language into machine code.

Assembly Language

A symbolic representation of the machine language of a specific processor, augmented by additional types of statements that facilitate program writing and that provide instructions to the assembler.

Compiler

A program that converts another program from some source language (or programming language) to machine language (object code). Some compilers output assembly language which is then converted to machine language by a separate assembler. A compiler is distinguished from an assembler by the fact that each input statement does not, in general, correspond to a single machine instruction or fixed sequence of instructions. A compiler may support such features as automatic allocation of variables, arbitrary arithmetic expressions, control structures such as FOR and WHILE loops, variable scope, input/output operations, higher-order functions and portability of source code.

Executable Code

The machine code generated by a source code language processor such as an assembler or compiler. This is software in a form that can be run in the computer.

Instruction Set

The collection of all possible instructions for a particular computer; that is, the collection of machine language instructions that a particular processor understands.

Linker

A utility program that combines one or more files containing object code from separately compiled program modules into a single file containing loadable or executable code.

Loader

A program routine that copies an executable program into memory for execution.

Machine Language, or Machine Code

The binary representation of a computer program which is actually read and interpreted by the computer. A program in machine code consists of a sequence of machine instructions (possibly interspersed with data). Instructions are binary strings which may be either all the same size (e.g., one 32-bit word for many modern RISC microprocessors) or of different sizes.

Object Code

The machine language representation of programming source code. Object code is created by a compiler or assembler and is then turned into executable code by the linker.

3. **Debugging and verifying.** Assembly code is more difficult to debug and verify because there are more possibilities for errors than in high-level code.
4. **Maintainability.** Assembly code is more difficult to modify and maintain because the language allows unstructured spaghetti code and all kinds of tricks that are difficult for others to understand. Thorough documentation and a consistent programming style are needed.
5. **Portability.** Assembly code is platform-specific. Porting to a different platform is difficult.

6. **System code can use intrinsic functions instead of assembly.** The best modern C++ compilers have intrinsic functions for accessing system control registers and other system instructions. Assembly code is no longer needed for device drivers and other system code when intrinsic functions are available.
7. **Application code can use intrinsic functions or vector classes instead of assembly.** The best modern C++ compilers have intrinsic functions for vector operations and other special instructions that previously required assembly programming.
8. **Compilers have been improved a lot in recent years.** The best compilers are now quite good. It takes a lot of expertise and experience to optimize better than the best C++ compiler.

Yet there are still some advantages to the occasional use of assembly language, including the following [FOG08a]:

1. **Debugging and verifying.** Looking at compiler-generated assembly code or the disassembly window in a debugger is useful for finding errors and for checking how well a compiler optimizes a particular piece of code.
2. **Making compilers.** Understanding assembly coding techniques is necessary for making compilers, debuggers, and other development tools.
3. **Embedded systems.** Small embedded systems have fewer resources than PCs and mainframes. Assembly programming can be necessary for optimizing code for speed or size in small embedded systems.
4. **Hardware drivers and system code.** Accessing hardware, system control registers, and so on may sometimes be difficult or impossible with high level code.
5. **Accessing instructions that are not accessible from high-level language.** Certain assembly instructions have no high-level language equivalent.
6. **Self-modifying code.** Self-modifying code is generally not profitable because it interferes with efficient code caching. It may, however, be advantageous, for example, to include a small compiler in math programs where a user-defined function has to be calculated many times.
7. **Optimizing code for size.** Storage space and memory is so cheap nowadays that it is not worth the effort to use assembly language for reducing code size. However, cache size is still such a critical resource that it may be useful in some cases to optimize a critical piece of code for size in order to make it fit into the code cache.
8. **Optimizing code for speed.** Modern C++ compilers generally optimize code quite well in most cases. But there are still cases where compilers perform poorly and where dramatic increases in speed can be achieved by careful assembly programming.
9. **Function libraries.** The total benefit of optimizing code is higher in function libraries that are used by many programmers.
10. **Making function libraries compatible with multiple compilers and operating systems.** It is possible to make library functions with multiple entries that are compatible with different compilers and different operating systems. This requires assembly programming.

The terms *assembly language* and *machine language* are sometimes, erroneously, used synonymously. Machine language consists of instructions directly executable by the processor. Each machine language instruction is a binary string containing an opcode, operand references, and perhaps other bits related to execution, such as flags. For convenience, instead of writing an instruction as a bit string, it can be written symbolically, with names for opcodes and registers. An assembly language makes much greater use of symbolic names, including assigning names to specific main memory locations and specific instruction locations. Assembly language also includes statements that are not directly executable but serve as instructions to the assembler that produces machine code from an assembly language program.

Assembly Language Elements

A statement in a typical assembly language has the form shown in Figure B.1. It consists of four elements: label, mnemonic, operand, and comment.

LABEL If a label is present, the assembler defines the label as equivalent to the address into which the first byte of the object code generated for that instruction will be loaded. The programmer may subsequently use the label as an address or as data in another instruction's address field. The assembler replaces the label with the assigned value when creating an object program. Labels are most frequently used in branch instructions.

As an example, here is a program fragment:

```
L2: SUB EAX, EDX   ;subtract contents of register EDX from
                   ;contents of EAX and store result in EAX
    JG  L2         ;jump to L2 if result of subtraction is
                   ;positive
```

The program will continue to loop back to location L2 until the result is zero or negative. Thus, when the jg instruction is executed, if the result is positive, the processor places the address equivalent to the label L2 in the program counter.

Reasons for using a label include the following;

1. A label makes a program location easier to find and remember.
2. The label can easily be moved to correct a program. The assembler will automatically change the address in all instructions that use the label when the program is reassembled.
3. The programmer does not have to calculate relative or absolute memory addresses, but just uses labels as needed.

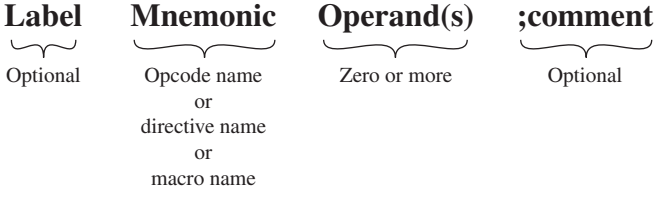


Figure B.1 Assembly-Language Statement Structure

MNEMONIC The mnemonic is the name of the operation or function of the assembly language statement. As discussed subsequently, a statement can correspond to a machine instruction, an assembler directive, or a macro. In the case of a machine instruction, a mnemonic is the symbolic name associated with a particular opcode.

Table 12.8 lists the mnemonic, or instruction name, of many of the x86 instructions. Appendix A of [CART06] lists the x86 instructions, together with the operands for each and the effect of the instruction on the condition codes. Appendix B of the NASM manual provides a more detailed description of each x86 instruction. Both documents are available at this book's Premium Content site.

OPERAND(S) An assembly language statement includes zero or more operands. Each operand identifies an immediate value, a register value, or a memory location. Typically, the assembly language provides conventions for distinguishing among the three types of operand references, as well as conventions for indicating addressing mode.

For the x86 architecture, an assembly language statement may refer to a register operand by name. Figure B.2 illustrates the general-purpose x86 registers, with their symbolic name and their bit encoding. The assembler will translate the symbolic name into the binary identifier for the register.

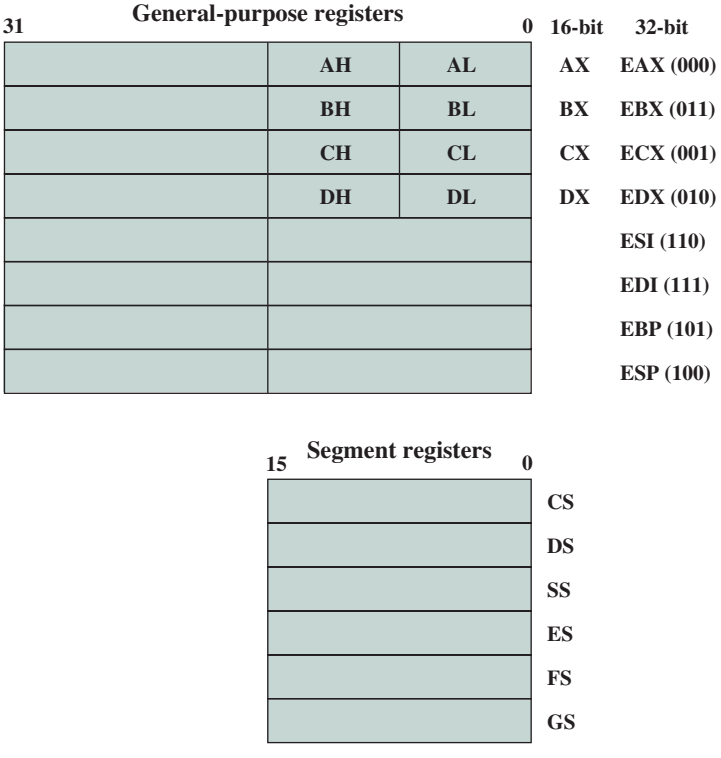


Figure B.2 Intel x86 Program Execution Registers

As discussed in Section 11.2, the x86 architecture has a rich set of addressing modes, each of which must be expressed symbolically in the assembly language. Here we cite a few of the common examples. For **register addressing**, the name of the register is used in the instruction. For example, `MOV ECX, EBX` copies the contents of register EBX into register ECX. Immediate addressing indicates that the value is encoded in the instruction. For example, `MOV EAX, 100H` copies the hexadecimal value 100 into register EAX. The immediate value can be expressed as a binary number with the suffix B or a decimal number with no suffix. Thus, equivalent statements to the preceding one are `MOV EAX, 100000000B` and `MOV EAX, 256`. **Direct addressing** refers to a memory location and is expressed as a displacement from the DS segment register. This is best explained by example. Assume that the 16-bit data segment register DS contains the value 1000H. Then the following sequence occurs:

```
MOV AX, 1234H
MOV [3518H], AX
```

First the 16-bit register AX is initialized to 1234H. Then, in line two, the contents of AX are moved to the logical address DS:3518H. This address is formed by shifting the contents of DS left 4 bits and adding 3518H to form the 32-bit logical address 13518H.

COMMENT All assembly languages allow the placement of comments in the program. A comment can either occur at the right-hand end of an assembly statement or can occupy an entire text line. In either case, the comment begins with a special character that signals to the assembler that the rest of the line is a comment and is to be ignored by the assembler. Typically, assembly languages for the x86 architecture use a semicolon (;) for the special character.

Type of Assembly Language Statements

Assembly language statements are one of four types: instruction, directive, macro definition, and comment. A comment statement is simply a statement that consists entirely of a comment. The remaining types are briefly described in this section.

INSTRUCTIONS The bulk of the noncomment statements in an assembly language program are symbolic representations of machine language instructions. Almost invariably, there is a one-to-one relationship between an assembly language instruction and a machine instruction. The assembler resolves any symbolic references and translates the assembly language instruction into the binary string that comprises the machine instruction.

DIRECTIVES Directives, also called **pseudo-instructions**, are assembly language statements that are not directly translated into machine language instructions. Instead, directives are instruction to the assembler to perform specified actions doing the assembly process. Examples include the following:

- Define constants
- Designate areas of memory for data storage
- Initialize areas of memory
- Place tables or other fixed data in memory
- Allow references to other programs

Table B.2 lists some of the NASM directives. As an example, consider the following sequence of statements:

Table B.2 Some NASM Assembly-Language Directives

(a) Letters for RES*x* and D*x* Directives

Unit	Letter
byte	B
word (2 bytes)	W
double word (4 bytes)	D
quad word (8 bytes)	Q
ten bytes	T

(b) Directives

Name	Description	Example
DB, DW, DD, DQ, DT	Initialize locations	`L6 DD 1A92H` `;doubleword at L6 initialized to 1A92H`
RESB, RESW, RESD, RESQ, REST	Reserve uninitialized locations	`BUFFER RESB 64` `;reserve 64 bytes starting at BUFFER`
INCBIN	Include binary file in output	`INCBIN "file.dat" ; include this file`
EQU	Define a symbol to a given constant value	`MSGLEN EQU 25` `;the constant MSGLEN equals decimal 25`
TIMES	Repeat instruction multiple times	`ZEROBUF TIMES 64 DB 0` `;initialize 64-byte buffer to all zeros`

```
L2 DB   "A"       ;byte initialized to ASCII code for A (65)
   MOV  AL, [L1]  ;copy byte at L1 into AL
   MOV  EAX, L1   ;store address of byte at L1 in EAX
   MOV  [L1], AH  ;copy contents of AH into byte at L1
```

If a plain label is used, it is interpreted as the address (or offset) of the data. If the label is placed inside square brackets, it is interpreted as the data at the address.

MACRO DEFINITIONS A macro definition is similar to a subroutine in several ways. A subroutine is a section of a program that is written once, and can be used multiple times by calling the subroutine from any point in the program. When a program is compiled or assembled, the subroutine is loaded only once. A call to the subroutine transfers control to the subroutine and a return instruction in the subroutine returns control to the point of the call. Similarly, a macro definition is a section of code that the programmer writes once, and then can use many times. The main difference is that when the assembler encounters a macro call, it replaces the macro call with the macro itself. This process is called **macro expansion**. So, if a macro is defined in an

assembly language program and invoked 10 times, then 10 instances of the macro will appear in the assembled code. In essence, subroutines are handled by the hardware at run time, whereas macros are handled by the assembler at assembly time. Macros provide the same advantage as subroutines in terms of modular programming, but without the runtime overhead of a subroutine call and return. The tradeoff is that the macro approach uses more space in the object code.

In NASM and many other assemblers, a distinction is made between a single-line macro and a multi-line macro. In NASM, single-line macros are defined using the %DEFINE directive. Here is an example in which multiple single-line macros are expanded. First, we define two macros:

```
%DEFINE B(X) = 2*X
%DEFINE A(X) = 1 + B(X)
```

At some point in the assembly language program, the following statement appears:

```
MOV AX, A(8)
```

The assembler expands this statement to:

```
MOV AX, 1+2*8
```

which assembles to a machine instruction to move the immediate value 17 to register AX.

Multiline macros are defined using the mnemonic %MACRO. Here is an example of a multiline macro definition:

```
%MACRO PROLOGUE 1
       PUSH EBP       ;push contents of EBP onto stack
                      ;pointed to by ESP and
                      ;decrement contents of ESP by 4
       MOV EBP, ESP   ;copy contents of ESP to EBP
       SUB ESP, %1    ;subtract first parameter value from ESP
```

The number 1 after the macro name in the `%MACRO` line defines the number of parameters the macro expects to receive. The use of %1 inside the macro definition refers to the first parameter to the macro call.

The macro call

```
MYFUNC: PROLOGUE 12
```

expands to the following lines of code:

```
MYFUNC: PUSH   EBP
        MOV    EBP, ESP
        SUB    ESP, 12
```

Example: Greatest Common Divisor Program

As an example of the use of assembly language, we look at a program to compute the greatest common divisor of two integers. We define the greatest common divisor of the integers a and b as follows:

$$\gcd(a,b) = \max[k, \text{such that } k \text{ divides } a \text{ and } k \text{ divides } b]$$

where we say that k divides a if there is no remainder. Euclid's algorithm for the greatest common divisor is based on the following theorem. For any nonnegative integers a and b,

$$\gcd(a,b) = \gcd(b, a \bmod b)$$

Here is a C language program that implements Euclid's algorithm:

```
unsigned int gcd (unsigned int a, unsigned int b)

{
      if (a == 0 && b == 0)
          b = 1;
      else if (b == 0)
          b = a;
      else if (a != 0)
          while (a != b)
             if (a < b)
                 b -= a;
             else
                 a -= b;
      return b;
}
```

Figure B.3 shows two assembly language versions of the preceding program. The program on the left was done by a C compiler; the program on the right was programmed by hand. The latter program uses a number of programmer's tricks to produce a tighter, more efficient implementation.

B.2 ASSEMBLERS

The **assembler** is a software utility that takes an assembly program as input and produces object code as output. The object code is a binary file. The assembler views this file as a block of memory starting at relative location 0.

There are two general approaches to assemblers: the two-pass assembler and the one-pass assembler.

Two-Pass Assembler

We look first at the two-pass assembler, which is more common and somewhat easier to understand. The assembler makes two passes through the source code (Figure B.4):

FIRST PASS In the first pass, the assembler is only concerned with label definitions. The first pass is used to construct a **symbol table** that contains a list of all labels and their associated **location counter** (LC) values. The first byte of the object code will have the LC value of 0. The first pass examines each assembly statement. Although the assembler is not yet ready to translate instructions, it must examine

```
gcd:    mov     ebx,eax
        mov     eax,edx
        test    ebx,ebx
        jne     L1
        test    edx,edx
        jne     L1
        mov     eax,1
        ret
L1:     test    eax,eax
        jne     L2
        mov     eax,ebx
        ret
L2:     test    ebx,ebx
        je      L5
L3:     cmp     ebx,eax
        je      L5
        jae     L4
        sub     eax,ebx
        jmp     L3
L4:     sub     ebx,eax
        jmp     L3
L5:     ret
```

(a) Compiled program

```
gcd:    neg     eax
        je      L3
L1:     neg     eax
        xchg    eax,edx
L2:     sub     eax,edx
        jg      L2
        jne     L1
L3:     add     eax,edx
        jne     L4
        inc     eax
L4:     ret
```

(b) Written directly in assembly language

Figure B.3 Assembly Programs for Greatest Common Divisor

each instruction sufficiently to determine the length of the corresponding machine instruction and therefore how much to increment the LC. This may require not only examining the opcode but also looking at the operands and the addressing modes.

Directives such as DQ and REST (see Table B.2) cause the location counter to be adjusted according to how much storage is specified.

When assembler encounters a statement with a label, it places the label into the symbol table, along with the current LC value. The assembler continues until it has read all of the assembly language statements.

SECOND PASS The second pass reads the program again from the beginning. Each instruction is translated into the appropriate binary machine code. Translation includes the following operations:

1. Translate the mnemonic into a binary opcode.
2. Use the opcode to determine the format of the instruction and the location and length of the various fields in the instruction.
3. Translate each operand name into the appropriate register or memory code.
4. Translate each immediate value into a binary string.
5. Translate any references to labels into the appropriate LC value using the symbol table.
6. Set any other bits in the instruction that are needed, including addressing mode indicators, condition code bits, and so on.

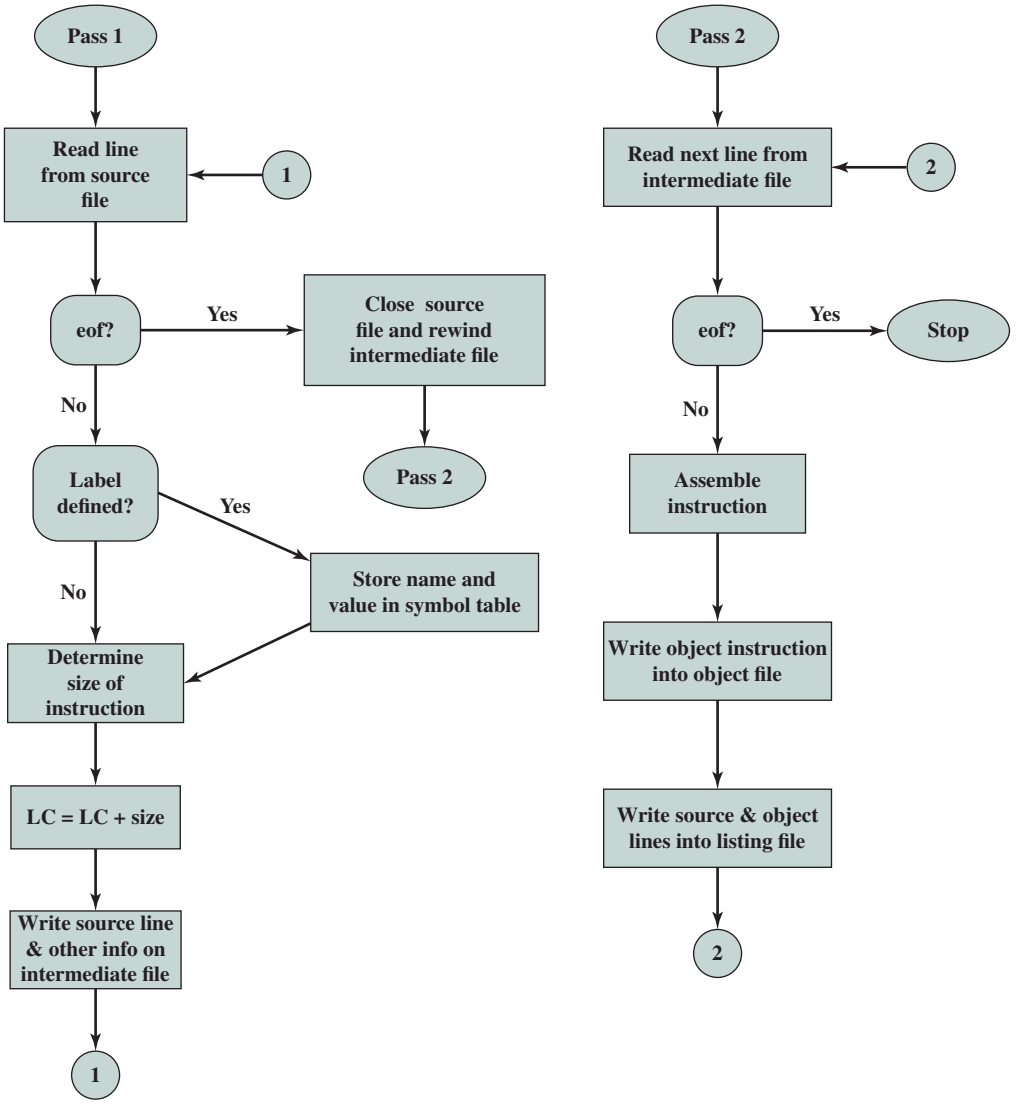


Figure B.4 Flowchart of Two-Pass Assembler

A simple example, using the ARM assembly language, is shown in Figure B.5. The ARM assembly language instruction ADDS r3, r3, #19 is translated in to the binary machine instruction 1110 0010 0101 0011 0011 0000 0001 0011.

ZEROTH PASS Most assembly language includes the ability to define macros. When macros are present there is an additional pass that the assembler must make before the first pass. Typically, the assembly language requires that all macro definitions must appear at the beginning of the program.

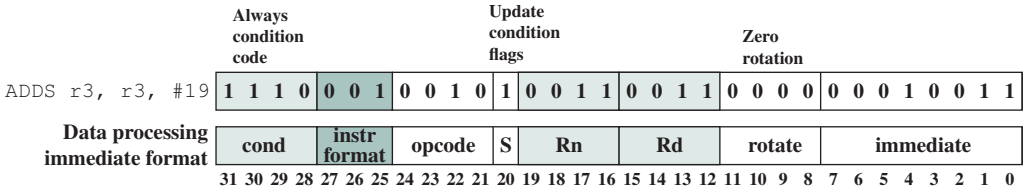

Always condition code
Update condition flags
Zero rotation
ADDS r3, r3, #19
1 1 1 0 0 0 1 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 0 0 0 1 0 0 1 1
Data processing immediate format
cond
instr format
opcode
S
Rn
Rd
rotate
immediate
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Figure B.5 Translating an ARM Assembly Instruction into a Binary Machine Instruction

The assembler begins this "zeroth pass" by reading all macro definitions. Once all the macros are recognized, the assembler goes through the source code and expands the macros with their associated parameters whenever a macro call is encountered. The macro processing pass generates a new version of the source code with all of the macro expansions in place and all of the macro definitions removed.

One-Pass Assembler

It is possible to implement an assembler that makes only a single pass through the source code (not counting the macro processing pass). The main difficulty in trying to assemble a program in one pass involves forward references to labels. Instruction operands may be symbols that have not yet been defined in the source program. Therefore, the assembler does not know what relative address to insert in the translated instruction.

In essence, the process of resolving forward references works as follows. When the assembler encounters an instruction operand that is a symbol that is not yet defined, the assembler does the following:

1. It leaves the instruction operand field empty (all zeros) in the assembled binary instruction.
2. The symbol used as an operand is entered in the symbol table. The table entry is flagged to indicate that the symbol is undefined.
3. The address of the operand field in the instruction that refers to the undefined symbol is added to a list of forward references associated with the symbol table entry.

When the symbol definition is encountered so that a LC value can be associated with it, the assembler inserts the LC value in the appropriate entry in the symbol table. If there is a forward reference list associated with the symbol, then the assembler inserts the proper address into any instruction previously generated that is on the forward reference list.

Example: Prime Number Program

We now look at an example that includes directives. This example looks at a program that finds prime numbers. Recall that prime numbers are evenly divisible by only 1 and themselves. There is no formula for doing this. The basic method this program uses is to find the factors of all odd numbers below a given limit. If no factor can be

```
unsigned guess;                          /* current guess for prime */
unsigned factor;                         /* possible factor of guess */
unsigned limit;                          /* find primes up to this value */

printf ("Find primes up to : ");
scanf("%u", &limit);
printf ("2\n");                          /* treat first two primes as */
printf ("3\n");                          /* special case */
guess = 5;                               /* initial guess */
while (guess < = limit) {                /* look for a factor of guess */
    factor = 3;
    while (factor * factor < guess && guess% factor != 0)
    factor + = 2;
    if (guess % factor != 0)
       printf ("%d\n", guess);
    guess += 2;                          /* only look at odd numbers */
}
```

Figure B.6 C Program for Testing Primality

found for an odd number, it is prime. Figure B.6 shows the basic algorithm written in C. Figure B.7 shows the same algorithm written in NASM assembly language.

B.3 LOADING AND LINKING

The first step in the creation of an active process is to load a program into main memory and create a process image (Figure B.8). Figure B.9 depicts a scenario typical for most systems. The application consists of a number of compiled or assembled modules in object-code form. These are linked to resolve any references between modules. At the same time, references to library routines are resolved. The library routines themselves may be incorporated into the program or referenced as shared code that must be supplied by the operating system at run time. In this section, we summarize the key features of linkers and loaders. First, we discuss the concept of relocation. Then, for clarity in the presentation, we describe the loading task when a single program module is involved; no linking is required. We can then look at the linking and loading functions as a whole.

Relocation

In a multiprogramming system, the available main memory is generally shared among a number of processes. Typically, it is not possible for the programmer to know in advance which other programs will be resident in main memory at the time of execution of his or her program. In addition, we would like to be able to swap active processes in and out of main memory to maximize processor utilization by providing a large pool of ready processes to execute. Once a program has been swapped out to disk, it would be quite limiting to declare that when it is next swapped back in, it must be placed in the same main memory region as before. Instead, we may need to **relocate** the process to a different area of memory.

```
%include "asm_io.inc"
segment .data
Message db "Find primes up to: ", 0

segment .bss
Limit resd 1                        ; find primes up to this limit
Guess resd 1                        ; the current guess for prime

segment .text
     global _asm_main
_asm_main:
     enter 0,0                      ; setup routine
     pusha

     mov eax, Message
     call print_string
     call read_int                  ; scanf("%u", & limit);
     mov [Limit], eax
     mov eax, 2                     ; printf("2\n");
     call print_int
     call print_nl
     mov eax, 3                     ; printf("3\n");
     call print_int
     call print_nl

     mov dword [Guess], 5           ; Guess = 5;
while_limit:                        ; while (Guess <= Limit)
     mov eax, [Guess]
     cmp eax, [Limit]
     jnbe end_while_limit           ; use jnbe since numbers are unsigned

     mov ebx, 3                     ; ebx is factor = 3;
while_factor:
     mov eax,ebx
     mul eax                        ; edx:eax = eax*eax
     jo end_while_factor            ; if answer won't fit in eax alone
     cmp eax, [Guess]
     jnb end_while_factor           ; if !(factor*factor < guess)
     mov eax,[Guess]
     mov edx,0
     div ebx                        ; edx = edx:eax% ebx
     cmp edx, 0
     je end_while_factor            ; if !(guess% factor != 0)

     add ebx,2; factor += 2;
     jmp while_factor
end_while_factor:
     je end_if                      ; if !(guess% factor != 0)
     mov eax,[Guess]                ; printf("%u\n")
     call print_int
     call print_nl
end_if:
     add dword [Guess], 2           ; guess += 2
     jmp while_limit
end_while_limit:

     popa
     mov eax, 0                     ; return back to C
     leave
     ret
```

Figure B.7 Assembly Program for Testing Primality

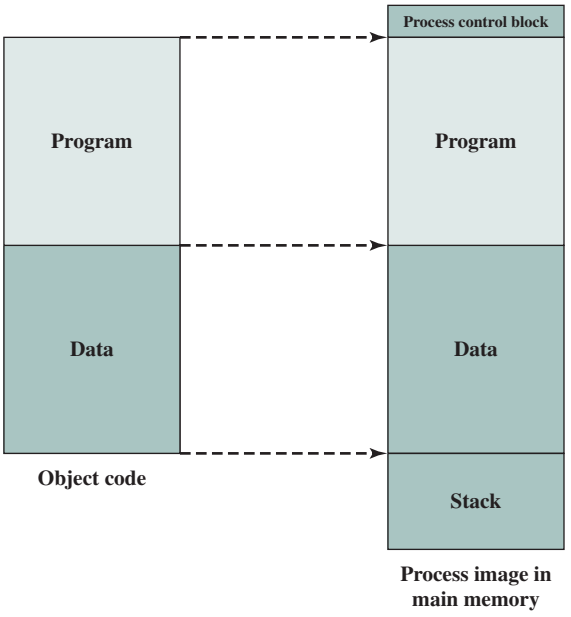


Figure B.8 The Loading Function

Thus, we cannot know ahead of time where a program will be placed, and we must allow that the program may be moved about in main memory due to swapping. These facts raise some technical concerns related to addressing, as illustrated in Figure B.10. The figure depicts a process image. For simplicity, let us assume that the process image occupies a contiguous region of main memory. Clearly, the

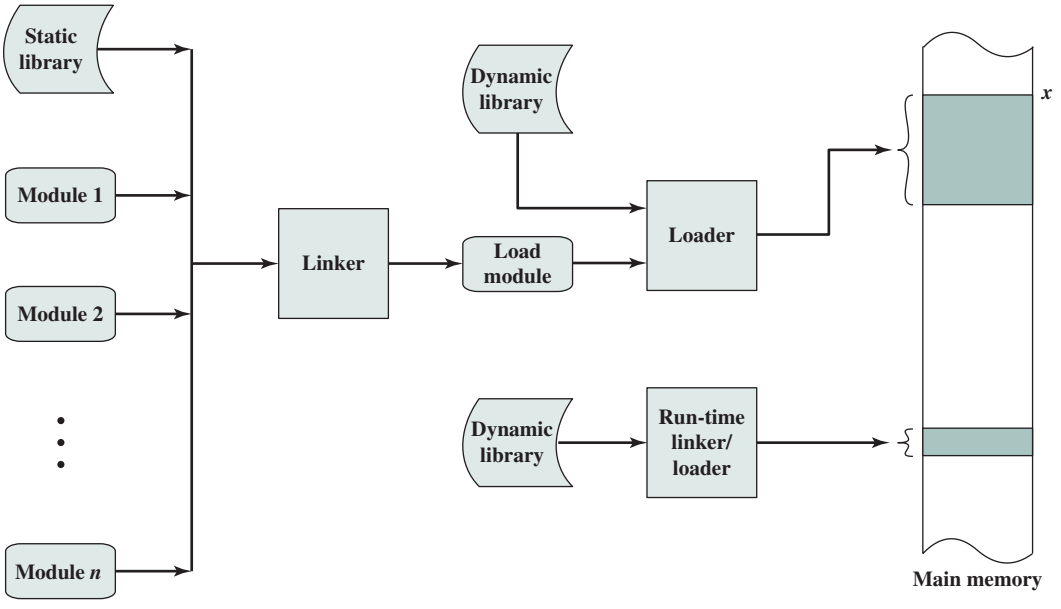


Figure B.9 A Linking and Loading Scenario

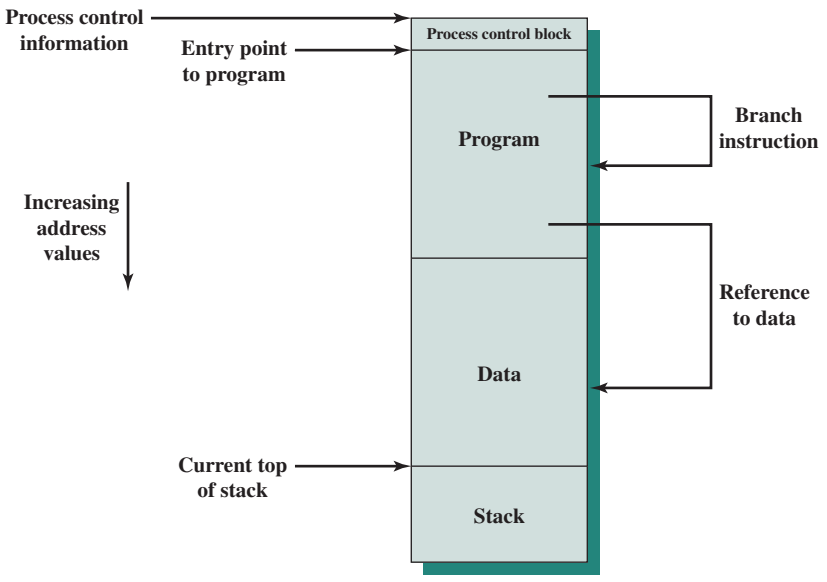


Figure B.10 Addressing Requirements for a Process

operating system will need to know the location of process control information and of the execution stack, as well as the entry point to begin execution of the program for this process. Because the operating system is managing memory and is responsible for bringing this process into main memory, these addresses are easy to come by. In addition, however, the processor must deal with memory references within the program. Branch instructions contain an address to reference the instruction to be executed next. Data reference instructions contain the address of the byte or word of data referenced. Somehow, the processor hardware and operating system software must be able to translate the memory references found in the code of the program into actual physical memory addresses, reflecting the current location of the program in main memory.

Loading

In Figure B.9, the loader places the load module in main memory starting at location x. In loading the program, the addressing requirement illustrated in Figure B.10 must be satisfied. In general, three approaches can be taken:

- Absolute loading
- Relocatable loading
- Dynamic run-time loading

ABSOLUTE LOADING An absolute loader requires that a given load module always be loaded into the same location in main memory. Thus, in the load module presented to the loader, all address references must be to specific, or absolute, main

memory addresses. For example, if x in Figure B.9 is location 1024, then the first word in a load module destined for that region of memory has address 1024.

The assignment of specific address values to memory references within a program can be done either by the programmer or at compile or assembly time (Table B.3a). There are several disadvantages to the former approach. First, every programmer would have to know the intended assignment strategy for placing modules into main memory. Second, if any modifications are made to the program that involve insertions or deletions in the body of the module, then all of the addresses will have to be altered. Accordingly, it is preferable to allow memory references within programs to be expressed symbolically and then resolve those symbolic references at the time of compilation or assembly. This is illustrated in Figure B.11. Every reference to an instruction or item of data is initially represented by a symbol. In preparing the module for input to an absolute loader, the assembler or compiler will convert all of these references to specific addresses (in this example, for a module to be loaded starting at location 1024), as shown in Figure B.11b.

Table B.3 Address Binding

(a) Loader

Binding Time	Function
Programming time	All actual physical addresses are directly specified by the programmer in the program itself.
Compile or assembly time	The program contains symbolic address references, and these are converted to actual physical addresses by the compiler or assembler.
Load time	The compiler or assembler produces relative addresses. The loader translates these to absolute addresses at the time of program loading.
Run time	The loaded program retains relative addresses. These are converted dynamically to absolute addresses by processor hardware.

(b) Linker

Linkage Time	Function
Programming time	No external program or data references are allowed. The programmer must place into the program the source code for all subprograms that are referenced.
Compile or assembly time	The assembler must fetch the source code of every subroutine that is referenced and assemble them as a unit.
Load module creation	All object modules have been assembled using relative addresses. These modules are linked together and all references are restated relative to the origin of the final load module.
Load time	External references are not resolved until the load module is to be loaded into main memory. At that time, referenced dynamic link modules are appended to the load module, and the entire package is loaded into main or virtual memory.
Run time	External references are not resolved until the external call is executed by the processor. At that time, the process is interrupted and the desired module is linked to the calling program.

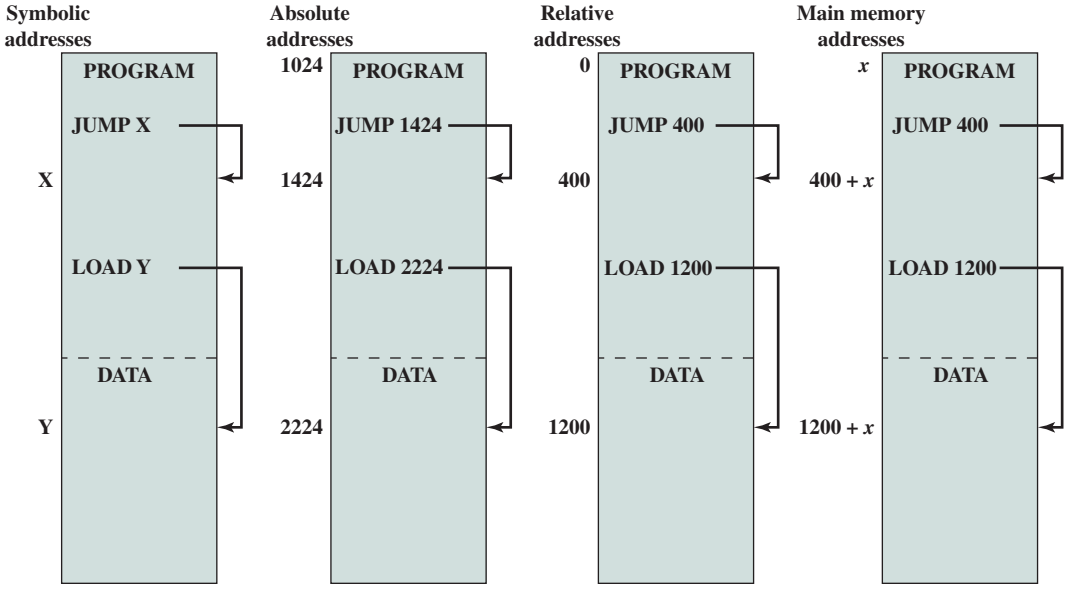


Figure B.11 Absolute and Relocatable Load Modules

RELOCATABLE LOADING The disadvantage of binding memory references to specific addresses prior to loading is that the resulting load module can only be placed in one region of main memory. However, when many programs share main memory, it may not be desirable to decide ahead of time into which region of memory a particular module should be loaded. It is better to make that decision at load time. Thus we need a load module that can be located anywhere in main memory.

To satisfy this new requirement, the assembler or compiler produces not actual main memory addresses (absolute addresses) but addresses that are relative to some known point, such as the start of the program. This technique is illustrated in Figure B.11c. The start of the load module is assigned the relative address 0, and all other memory references within the module are expressed relative to the beginning of the module.

With all memory references expressed in relative format, it becomes a simple task for the loader to place the module in the desired location. If the module is to be loaded beginning at location x, then the loader must simply add x to each memory reference as it loads the module into memory. To assist in this task, the load module must include information that tells the loader where the address references are and how they are to be interpreted (usually relative to the program origin, but also possibly relative to some other point in the program, such as the current location). This set of information is prepared by the compiler or assembler and is usually referred to as the relocation dictionary.

DYNAMIC RUN-TIME LOADING Relocatable loaders are common and provide obvious benefits relative to absolute loaders. However, in a multiprogramming

environment, even one that does not depend on virtual memory, the relocatable loading scheme is inadequate. We have referred to the need to swap process images in and out of main memory to maximize the utilization of the processor. To maximize main memory utilization, we would like to be able to swap the process image back into different locations at different times. Thus, a program, once loaded, may be swapped out to disk and then swapped back in at a different location. This would be impossible if memory references had been bound to absolute addresses at the initial load time.

The alternative is to defer the calculation of an absolute address until it is actually needed at run time. For this purpose, the load module is loaded into main memory with all memory references in relative form (Figure B.11c). It is not until an instruction is actually executed that the absolute address is calculated. To assure that this function does not degrade performance, it must be done by special processor hardware rather than software. This hardware is described in Chapter 8.

Dynamic address calculation provides complete flexibility. A program can be loaded into any region of main memory. Subsequently, the execution of the program can be interrupted and the program can be swapped out of main memory, to be later swapped back in at a different location.

Linking

The function of a linker is to take as input a collection of object modules and produce a load module, consisting of an integrated set of program and data modules, to be passed to the loader. In each object module, there may be address references to locations in other modules. Each such reference can only be expressed symbolically in an unlinked object module. The linker creates a single load module that is the contiguous joining of all of the object modules. Each intramodule reference must be changed from a symbolic address to a reference to a location within the overall load module. For example, module A in Figure B.12a contains a procedure invocation of module B. When these modules are combined in the load module, this symbolic reference to module B is changed to a specific reference to the location of the entry point of B within the load module.

LINKAGE EDITOR The nature of this address linkage will depend on the type of load module to be created and when the linkage occurs (Table B.3b). If, as is usually the case, a relocatable load module is desired, then linkage is usually done in the following fashion. Each compiled or assembled object module is created with references relative to the beginning of the object module. All of these modules are put together into a single relocatable load module with all references relative to the origin of the load module. This module can be used as input for relocatable loading or dynamic run-time loading.

A linker that produces a relocatable load module is often referred to as a linkage editor. Figure B.12 illustrates the linkage editor function.

DYNAMIC LINKER As with loading, it is possible to defer some linkage functions. The term *dynamic linking* is used to refer to the practice of deferring the linkage of some external modules until after the load module has been created. Thus, the load module contains unresolved references to other programs. These references can be resolved either at load time or run time.

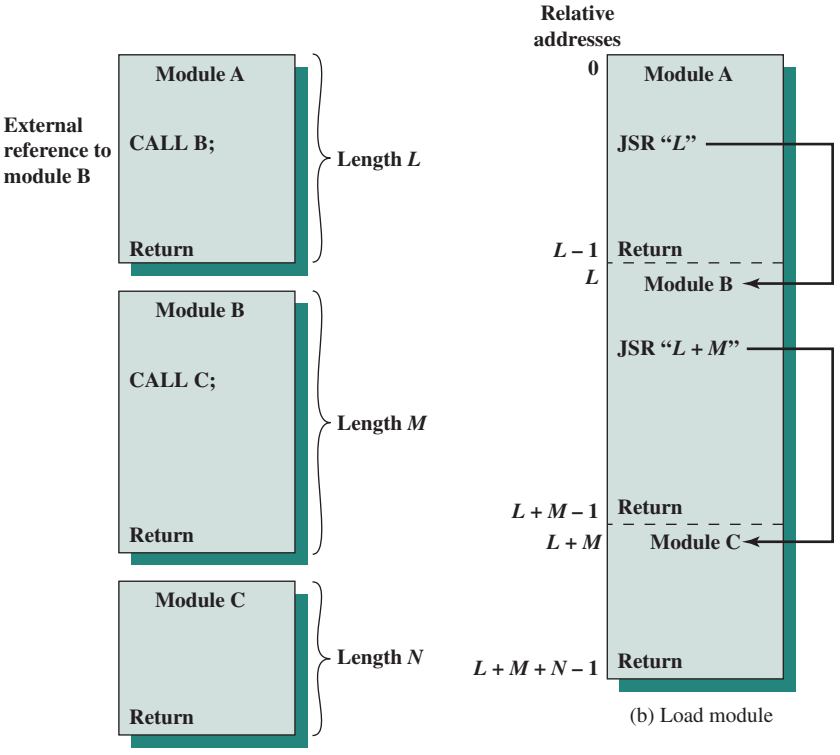


Figure B.12 The Linking Function

For **load-time dynamic linking** (involving upper dynamic library in Figure B.9), the following steps occur. The load module (application module) to be loaded is read into memory. Any reference to an external module (target module) causes the loader to find the target module, load it, and alter the reference to a relative address in memory from the beginning of the application module. There are several advantages to this approach over what might be called static linking:

- It becomes easier to incorporate changed or upgraded versions of the target module, which may be an operating system utility or some other general-purpose routine. With static linking, a change to such a supporting module would require the relinking of the entire application module. Not only is this inefficient, but it may be impossible in some circumstances. For example, in the personal computer field, most commercial software is released in load module form; source and object versions are not released.
- Having target code in a dynamic link file paves the way for automatic code sharing. The operating system can recognize that more than one application is using the same target code because it loaded and linked that code. It can use that information to load a single copy of the target code and link it to both applications, rather than having to load one copy for each application.

- It becomes easier for independent software developers to extend the functionality of a widely used operating system such as Linux. A developer can come up with a new function that may be useful to a variety of applications and package it as a dynamic link module.

With **run-time dynamic linking** (involving lower dynamic library in Figure B.9), some of the linking is postponed until execution time. External references to target modules remain in the loaded program. When a call is made to the absent module, the operating system locates the module, loads it, and links it to the calling module. Such modules are typically shareable. In the Windows environment, these are called dynamic-link libraries (DLLs) Thus, if one process is already making use of a dynamically linked shared module, then that module is in main memory and a new process can simply link to the already-loaded module.

The use of DLLs can lead to a problem commonly referred to as **DLL hell**. DLL hell occurs if two or more processes are sharing a DLL module but expect different versions of the module. For example, an application or system function might be re-installed and bring in with it an older version of a DLL file.

We have seen that dynamic loading allows an entire load module to be moved around; however, the structure of the module is static, being unchanged throughout the execution of the process and from one execution to the next. However, in some cases, it is not possible to determine prior to execution which object modules will be required. This situation is typified by transaction-processing applications, such as an airline reservation system or a banking application. The nature of the transaction dictates which program modules are required, and they are loaded as appropriate and linked with the main program. The advantage of the use of such a dynamic linker is that it is not necessary to allocate memory for program units unless those units are referenced. This capability is used in support of segmentation systems.

One additional refinement is possible: An application need not know the names of all the modules or entry points that may be called. For example, a charting program may be written to work with a variety of plotters, each of which is driven by a different driver package. The application can learn the name of the plotter that is currently installed on the system from another process or by looking it up in a configuration file. This allows the user of the application to install a new plotter that did not exist at the time the application was written.

B.4 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

Assembler	label	mnemonic
assembly language	linkage editor	one-pass assembler
comment	linking	operand
directive	load-time dynamic linking	relocation
dynamic linker	loading	run-time dynamic linking
instruction	macro	two-pass assembler

Review Questions

B.1 List some reasons why it is worthwhile to study assembly language programming.
B.2 What is an assembly language?
B.3 List some disadvantages of assembly language compared to high-level languages.
B.4 List some advantages of assembly language compared to high-level languages.
B.5 What are the typical elements of an assembly language statement?
B.6 List and briefly define four different kinds of assembly language statements.
B.7 What is the difference between a one-pass assembler and a two-pass assembler?

Problems

B.1 Core War is a programming game introduced to the public in the early 1980s [DEWD84], which was popular for a period of 15 years or so. Core War has four main components: a memory array of 8000 addresses; a simplified assembly language Redcode; an executive program called MARS (an acronym for Memory Array Redcode Simulator); and the set of contending battle programs. Two battle programs are entered into the memory array at randomly chosen positions; neither program knows where the other one is. MARS executes the programs in a simple version of time-sharing. The two programs take turns; a single instruction of the first program is executed, then a single instruction of the second, and so on. What a battle program does during the execution cycles allotted to it is entirely up to the programmer. The aim is to destroy the other program by ruining its instructions. In this problem and the next several, we use an even simpler language, called CodeBlue, to explore some Core War concepts.

CodeBlue contains only five assembly language statements and uses three addressing modes (Table B.4). Addresses wrap around, so that for the last location in memory, the relative address of +1 refers to the first location in memory. For example, `ADD #4, 6` adds 4 to the contents of relative location 6 and stores the results in location 6; `JUMP @5` transfers execution to the memory address contained in the location five slots past the location of the current `JUMP` instruction.

a. The program Imp is the single instruction `COPY 0, 1`. What does it do?
b. The program Dwarf is the following sequence of instructions:

```
ADD #4, 3
COPY 2, @2
JUMP -2
DATA 0
```

What does it do?
c. Rewrite Dwarf using symbols, so that it looks more like a typical assembly language program.

B.2 What happens if we pit Imp against Dwarf?
B.3 Write a "carpet bombing" program in CodeBlue that zeros out all of memory (with the possible exception of the program locations).
B.4 How would the previous program fare against Imp?
B.5 **a.** What is the value of the C status flag after the following sequence:

```
mov al, 3
add al, 4
```

b. What is the value of the C status flag after the following sequence:

```
mov al, 3
sub al, 4
```

Table B.4 CodeBlue Assembly Language

(a) Instruction Set

Format		Meaning
DATA	<value>	<value> set at current location
COPY	A, B	copies source A to destination B
ADD	A, B	adds A to B, putting result in B
JUMP	A	transfer execution to A
JUMPZ	A, B	if B = 0, transfer to A

(b) Addressing Modes

Mode	Format	Meaning
Literal	# followed by value	This is an immediate mode, the operand value is in the instruction.
Relative	Value	The value represents an offset from the current location, which contains the operand.
Indirect	@ followed by value	The value represents an offset from the current location; the offset location contains the relative address of the location that contains the operand.

```
Loop COPY #0, -1
     JUMP -1
```

Hint: Remember that instruction execution alternates between the two opposing programs.

B.6 Consider the following NAMS instruction:

```
cmp vleft, vright
```

For signed integers, there are three status flags that are relevant. If vleft = vright, then ZF is set. If vleft > vright, ZF is unset (set to 0) and SF = OF. If vleft < vright, ZF is unset and SF ≠ OF. Why does SF = OF if vleft > vright?

B.7 Consider the following NASM code fragment:

```
mov al, 0
cmp al, al
je next
```

Write an equivalent program consisting of a single instruction.

B.8 Consider the following C program:

```
/* a simple C program to average 3 integers */
main ()
{ int avg;
  int i1 = 20;
  int i2 = 13;
  int i3 = 82;
  avg = (i1 + i2 + i3)/3;
}
```

Write an NASM version of this program.

B.9 Consider the following C code fragment:

```
if (EAX == 0) EBX = 1;
else EBX = 2;
```

Write an equivalent NASM code fragment.

B.10 The initialize data directives can be used to initialize multiple locations. For example,

```
db 0x55,0x56,0x57
```

reserves three bytes and initializes their values.

NASM supports the special token $ to allow calculations to involve the current assembly position. That is, $ evaluates to the assembly position at the beginning of the line containing the expression. With the preceding two facts in mind, consider the following sequence of directives:

```
message db 'hello, world'
msglen equ $-message
```

What value is assigned to the symbol msglen?

B.11 Assume the three symbolic variables V1, V2, V3 contain integer values. Write an NASM code fragment that moves the smallest value into integer ax. Use only the instructions `mov`, `cmp`, and `jbe`.

B.12 Describe the effect of this instruction: `cmp eax, 1` Assume that the immediately preceding instruction updated the contents of eax.

B.13 The `xchg` instruction can be used to exchange the contents of two registers. Suppose that the x86 instruction set did not support this instruction.

a. Implement `xchg ax, bx` using only push and pop instructions.

b. Implement `xchg ax, bx` using only the `xor` instruction (do not involve other registers).

B.14 In the following program, assume that a, b, x, y are symbols for main memory locations. What does the program do? You can answer the question by writing the equivalent logic in C.

```
        mov     eax,a
        mov     ebx,b
        xor     eax,x
        xor     ebx,y
        or      eax,ebx
        jnz     L2
L1:                     ;sequence of instructions…
        jmp     L3
L2:                     ;another sequence of instructions…
L3:
```

B.15 Section B.1 includes a C program that calculates the greatest common divisor of two integers.

a. Describe the algorithm in words and show how the program does implement the Euclid algorithm approach to calculating the greatest common divisor.

b. Add comments to the assembly program of Figure B.3a to clarify that it implements the same logic as the C program.

c. Repeat part (b) for the program of Figure B.3b.

B.16 **a.** A 2-pass assembler can handle future symbols and an instruction can therefore use a future symbol as an operand. This is not always true for directives. The EQU directive, for example, cannot use a future symbol. The directive "A EQU B + 1" is easy to execute if B is previously defined, but impossible if B is a future symbol. What's the reason for this?

b. Suggest a way for the assembler to eliminate this limitation such that any source line could use future symbols.

B.17 Consider a symbol directive MAX of the following form: symbol MAX list of expressions

The label is mandatory and is assigned the value of the largest expression in the operand field. Example:

```
MSGLEN MAX A, B, C ;where A, B, C are defined symbols
```

How is MAX executed by the Assembler and in what pass?

REFERENCES

ABBREVIATIONS

ACM	Association for Computing Machinery
IEEE	Institute of Electrical and Electronics Engineers
NIST	National Institute of Standards and Technology

AGAR89 Agarwal, A. *Analysis of Cache Performance for Operating Systems and Multiprogramming*. Boston: Kluwer Academic Publishers, 1989.

AGER87 Agerwala, T., and Cocke, J. *High Performance Reduced Instruction Set Processors.* Technical Report RC12434 (#55845). Yorktown, NY: IBM Thomas J. Watson Research Center, January 1987.

ALLA13 Allan, G. "DDR4 Bank Groups in Embedded Applications." *Chip Design*, August 26, 2013. chipdesignmag.com

ALTS12 Alschuler, F., and Gallmeier, J. "Heterogeneous System Architecture: Multicore Image Processing Use a Mix of CPU and GPU Elements." *Embedded Computing Design*, December 6, 2012.

AMDA67 Amdahl, G. "Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capability." *Proceedings of the AFIPS Conference*, 1967.

AMDA13 Amdahl, G. "Computer Architecture and Amdahl's Law." *Computer*, December 2013.

ANDE67a Anderson, D., Sparacio, F., and Tomasulo, F. "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling." *IBM Journal of Research and Development*, January 1967.

ANDE67b Anderson, S., et al. "The IBM System/360 Model 91: Floating-Point Execution Unit." *IBM Journal of Research and Development*, January 1967. Reprinted in [SWAR90, Volume 1].

ANTH08 Anthes, G. "What's Next for the x86?" *ComputerWorld*, June 16, 2008.

AROR12 Arora, M., et al. "Redefining the Role of the CPU in the Era of CPU-GPU Integration." *IEEE Micro*, November/December 2012.

ATKI96 Atkins, M. "PC Software Performance Tuning." *IEEE Computer*, August 1996.

AZIM92 Azimi, M., Prasad, B., and Bhat, K. "Two Level Cache Architectures." *Proceedings, COMPCON '92*, February 1992.

BACO94 Bacon, F., Graham, S., and Sharp, O. "Compiler Transformations for High-Performance Computing." *ACM Computing Surveys*, December 1994.

BAIL93 Bailey, D. "RISC Microprocessors and Scientific Computing." *Proceedings*, *Supercomputing'93*, 1993.

BELL70 Bell, C., Cady, R., McFarland, H., Delagi, B., O'Loughlin, J., and Noonan, R. "A New Architecture for Minicomputers—The DEC PDP-11." *Proceedings*, *Spring Joint Computer Conference*, 1970.

BELL71 Bell, C., and Newell, A. *Computer Structures: Readings and Examples.* New York: McGraw-Hill, 1971.

BELL78a Bell, C., Mudge, J., and McNamara, J. *Computer Engineering: A DEC View of Hardware Systems Design.* Bedford, MA: Digital Press, 1978.

BELL78b Bell, C., Newell, A., and Siewiorek, D. "Structural Levels of the PDP-8." In [BELL78a].

BELL78c Bell, C., Kotok, A., Hastings, T., and Hill, R. "The Evolution of the DEC System-10." *Communications of the ACM*, January 1978.

BENH92 Benham, J. "A Geometric Approach to Presenting Computer Representations of Integers." *SIGCSE Bulletin*, December 1992.

BOOT51 Booth, A. "A Signed Binary Multiplication Technique." *The Quarterly Journal of Mechanics and Applied Mathematics.* Vol. 4, No. 2, 1951.

BORK03 Borkar, S. "Getting Gigascale Chips: Challenges and Opportunities in Continuing Moore's Law." *ACM Queue*, October 2003.

BRAD91a Bradlee, D., Eggers, S., and Henry, R. "The Effect on RISC Performance of Register Set Size and Structure versus Code Generation Strategy." *Proceedings*, *18th Annual International Symposium on Computer Architecture*, May 1991.

BRAD91b Bradlee, D., Eggers, S., and Henry, R. "Integrating Register Allocation and Instruction Scheduling for RISCs." *Proceedings*, *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

BREW97 Brewer, E. "Clustering: Multiply and Conquer." *Data Communications*, July 1997.

BURG97 Burger, D., and Austin, T. "The SimpleScalar Tool Set, Version 2.0." *Computer Architecture News*, June 1997.

BURK46 Burks, A., Goldstine, H., and von Neumann, J. *Preliminary Discussion of the Logical Design of an Electronic Computer Instrument.* Report prepared for U.S. Army Ordnance Department, 1946, reprinted in [BELL71].

BUYY99 Buyya, R. *High Performance Cluster Computing: Architectures and Systems.* Upper Saddle River, NJ: Prentice Hall, 1999.

CANT01 Cantin, J., and Hill, H. "Cache Performance for Selected SPEC CPU2000 Benchmarks." *Computer Architecture News*, September 2001.

CART06 Carter, P. *PC Assembly Language.* July 23, 2006. http://www.drpaulcarter.com/pcasm/.

CEKL97 Cekleov, M., and Dubois, M. "Virtual-Address Caches, Part 1: Problems and Solutions in Uniprocessors." *IEEE Micro*, September/October 1997.

CHAI82 Chaitin, G. "Register Allocation and Spilling via Graph Coloring." *Proceedings*, *SIGPLAN Symposium on Compiler Construction*, June 1982.

CHOW86 Chow, F., Himmelstein, M., Killian, E., and Weber, L. "Engineering a RISC Compiler System." *Proceedings*, *COMPCON Spring '86*, March 1986.

CHOW87 Chow, F., Correll, S., Himmelstein, M., Killian, E., and Weber, L. "How Many Addressing Modes Are Enough?" *Proceedings*, *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987.

CHOW90 Chow, F., and Hennessy, J. "The Priority-Based Coloring Approach to Register Allocation." *ACM Transactions on Programming Languages*, October 1990.

CITR06 Citron, D., Hurani, A., and Gnadrey, A. "The Harmonic or Geometric Mean: Does it Really Matter?" *Computer Architecture News*, September 2006.

CLAR85 Clark, D., and Emer, J. "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement." *ACM Transactions on Computer Systems*, February 1985.

COHE81 Cohen, D. "On Holy Wars and a Plea for Peace." *Computer*, October 1981.

COOK82 Cook, R., and Dande, N. "An Experiment to Improve Operand Addressing." *Proceedings*, *Symposium on Architecture Support for Programming Languages and Operating Systems*, March 1982.

COLW85a Colwell, R., Hitchcock, C., Jensen, E., Brinkley-Sprunt, H., and Kollar, C. "Computers, Complexity, and Controversy." *Computer*, September 1985.

COLW85b Colwell, R., Hitchcock, C., Jensen, E., Brinkley-Sprunt, H., and Kollar, C. "More Controversy About 'Computers, Complexity, and Controversy.'" *Computer*, December 1985.

COON81 Coonen, J. "Underflow and Denormalized Numbers." *IEEE Computer*, March 1981.

COUT86 Coutant, D., Hammond, C., and Kelley, J. "Compilers for the New Generation of Hewlett-Packard Computers." *Proceedings*, *COMPCON Spring '86*, March 1986.

CRAG79 Cragon, H. "An Evaluation of Code Space Requirements and Performance of Various Architectures." *Computer Architecture News*, February 1979.

CRAW90 Crawford, J. "The i486 CPU: Executing Instructions in One Clock Cycle." *IEEE Micro*, February 1990.

CURR11 Curran, B., et al. "The zEnterprise 196 System and Microprocessor." *IEEE Micro*, March/April 2011.

DATT93 Dattatreya, G. "A Systematic Approach to Teaching Binary Arithmetic in a First Course." *IEEE Transactions on Education*, February 1993.

DAVI87 Davidson, J., and Vaughan, R. "The Effect of Instruction Set Complexity on Program Size and Memory Performance." *Proceedings, Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987.

DENN68 Denning, P. "The Working Set Model for Program Behavior." *Communications of the ACM*, May 1968.

DERO87 DeRosa, J., and Levy, H. "An Evaluation of Branch Architectures." *Proceedings, Fourteenth Annual International Symposium on Computer Architecture*, 1987.

DEWA90 Dewar, R., and Smosna, M. *Microprocessors: A Programmer's View.* New York: McGraw-Hill, 1990.

DEWD84 Dewdney, A. "In the Game Called Core War Hostile Programs Engage in a Battle of Bits." *Scientific American*, May 1984.

DOBO13 Dobos, I., et al. *IBM zEnterprise EC12 Technical Guide.* IBM Redbook SG24-8049-01, December 2013.

DOWD98 Dowd, K., and Severance, C. *High Performance Computing.* Sebastopol, CA: O'Reilly, 1998.

EISC07 Eischen, C. "RAID 6 Covers More Bases." *Network World*, April 9, 2007.

ELAY85 El-Ayat, K., and Agarwal, R. "The Intel 80386—Architecture and Implementation." *IEEE Micro*, December 1985.

FATA08 Fatahalian, K., and Houston, M. "A Closer Look at GPUs." *Communications of the ACM*, October 2008.

FEIT15 Feitelson, D. *Workload Modeling for Computer Systems Performance Evaluation.* Cambridge, UK: Cambridge University Press, 2015.

FLEM86 Fleming, P., and Wallace, J. "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results." *Communications of the ACM*, March 1986.

FLYN72 Flynn, M. "Some Computer Organizations and Their Effectiveness." *IEEE Transactions on Computers*, September 1972.

FLYN87 Flynn, M., Mitchell, C., and Mulder, J. "And Now a Case for More Complex Instruction Sets." *Computer*, September 1987.

FOG08 Fog, A. *Optimizing Subroutines in Assembly Language: An Optimization Guide for x86 Platforms.* Copenhagen University College of Engineering, 2008. http://www.agner.org/optimize/

FRAI83 Frailey, D. "Word Length of a Computer Architecture: Definitions and Applications." *Computer Architecture News*, June 1983.

GENU04 Genu, P. *A Cache Primer.* Application Note AN2663. Freescale Semiconductor, Inc., 2004. (available in Premium Content Document section)

GHAI98 Ghai, S., Joyner, J., and John, L. *Investigating the Effectiveness of a Third Level Cache.* Technical Report TR-980501-01, Laboratory for Computer Architecture, University of Texas at Austin, 1998.

GIBB04 Gibbs, W. "A Split at the Core." *Scientific American*, November 2004.

GIFF87 Gifford, D., and Spector, A. "Case Study: IBM's System/360-370 Architecture." *Communications of the ACM*, April 1987.

GILA95 Giladi, R., and Ahituv, N. "SPEC as a Performance Evaluation Measure." *Computer*, August 1995.

GOER12 Goering, R. "New Memory Technologies Challenge NAND Flash and DRAM." *Cadence Industry Insight Blogs*, August 22, 2012. http://community.cadence.com/cadence_blogs_8/b/ii/archive/2012/08/22/keynote-new-memory-technologies-challenge-nand-flash-and-dram

GOLD54 Goldstine, H., Pomerene, J., and Smith, C. *Final Progress Report on the Physical Realization of an Electronic Computing Instrument.* Princeton: The Institute for Advanced Study Electronic Computer Project, 1954.

GSOE08 Gsoedl, J. "Solid State: New Frontier in Storage." *Storage*, July 2008.

GUST88 Gustafson, J. "Reevaluating Amdahl's Law." *Communications of the ACM*, May 1988.

HAND98 Handy, J. *The Cache Memory Book.* San Diego: Academic Press, 1998.

HARR06 Harris, W. "Multi-Core in the Source Engine." *bit-tech.net technical paper*, November 2, 2006.

HAYE98 Hayes, J. *Computer Architecture and Organization.* New York: McGraw-Hill, 1998.

HEAT84 Heath, J. "Re-Evaluation of RISC 1." *Computer Architecture News*, March 1984.

HENN07 Henning, J. "SPEC CPU Suite Growth: An Historical Perspective." *Computer Architecture News*, March 2007.

HENN12 Hennessy, J., and Patterson, D. *Computer Architecture: A Quantitative Approach.* Waltham, MA: Morgan Kaufman, 2012.

HENN82 Hennessy, J., et al. "Hardware/Software Tradeoffs for Increased Performance." *Proceedings, Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982.

HENN84 Hennessy, J. "VLSI Processor Architecture." *IEEE Transactions on Computers*, December 1984.

HILL64 Hill, R. "Stored Logic Programming and Applications." *Datamation,* February 1964.

HILL89 Hill, M. "Evaluating Associativity in CPU Caches." *IEEE Transactions on Computers*, December 1989.

HUCK83 Huck, T. *Comparative Analysis of Computer Architectures.* Stanford University Technical Report No. 83-243, May 1983.

HUGG05 Huggahalli, R., Iyer, R., and Tetrick, S. "Direct Cache Access for High Bandwidth Network I/O." *Proceedings, 32nd Annual International Symposium on Computer Architecture*, 2005.

HUGU91 Huguet, M., and Lang, T. "Architectural Support for Reduced Register Saving/Restoring in Single-Window Register Files." *ACM Transactions on Computer Systems*, February 1991.

HWAN93 Hwang, K. *Advanced Computer Architecture.* New York: McGraw-Hill, 1993.

HWAN99 Hwang, K, et al. "Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space." *IEEE Concurrency*, January–March 1999.

INTE98 Intel Corp. *Pentium Pro and Pentium II Processors and Related Products.* Aurora, CO, 1998.

INTE04 Intel Research and Development. *Architecting the Era of Tera.* Intel White Paper, February 2004.

INTE08 Intel Corp. *Integrated Network Acceleration Features of Intel I/O Acceleration Technology and Microsoft Windows Server 2008.* Intel White Paper, February 2004.

INTE12 Intel Corp. *Intel Data Direct I/O Technology (Intel DDIO): A Primer.* Intel White Paper, February 2012.

INTE14 Intel Corp. *The Computer Architecture of Intel Processor Graphics Gen8.* Intel White Paper, September 2014.

ITRS14 *The International Technology Roadmap For Semiconductors, 2013 Edition*, 2014. http://www.itrs.net

JACO95 Jacob, B., and Mudge, T. "Notes on Calculating Computer Performance." *University of Michigan Tech Report CSE-TR-231-95*, March 1995.

JACO08 Jacob, B., Ng, S., and Wang, D. *Memory Systems: Cache, DRAM, Disk.* Boston: Morgan Kaufmann, 2008.

JAIN91 Jain, R. *The Art of Computer System Performance Analysis.* New York: Wiley, 1991.

JAME90 James, D. "Multiplexed Buses: The Endian Wars Continue." *IEEE Micro*, September 1983.

JEFF12 Jeff, B. *Advances in big.LITTLE Technology for Power and Energy Savings.* ARM White Paper, September 2012.

JOHN91 Johnson, M. *Superscalar Microprocessor Design.* Englewood Cliffs, NJ: Prentice Hall, 1991.

JOHN04 John, L. "More on finding a Single Number to indicate Overall Performance of a Benchmark Suite." *Computer Architecture News*, March 2004.

JOUP88 Jouppi, N. "Superscalar versus Superpipelined Machines." *Computer Architecture News*, June 1988.

JOUP89a Jouppi, N., and Wall, D. "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines." *Proceedings*, *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

JOUP89b Jouppi, N. "The Nonuniform Distribution of Instruction-Level and Machine Parallelism and its Effect on Performance." *IEEE Transactions on Computers*, December 1989.

KAPP00 Kapp, C. "Managing Cluster Computers." *Dr. Dobb's Journal*, July 2000.

KATE83 Katevenis, M. *Reduced Instruction Set Computer Architectures for VLSI.* Ph.D. Dissertation, Computer Science Department, University of California at Berkeley, October 1983. Reprinted by MIT Press, Cambridge, MA, 1985.

KATZ89 Katz, R., Gibson, G., and Patterson, D. "Disk System Architecture for High Performance Computing." *Proceedings of the IEEE*, December 1989.

KNUT71 Knuth, D. "An Empirical Study of FORTRAN Programs." *Software Practice and Experience*, Vol. 1, 1971.

KUCK77 Kuck, D., Parker, D., and Sameh, A. "An Analysis of Rounding Methods in Floating-Point Arithmetic." *IEEE Transactions on Computers*, July 1977.

KULT13 Kulrursay, E., et al. "Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative." *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

KUMA07 Kumar, A., and Huggahalli, R. "Impact of Cache Coherence Protocols on the Processing of Network Traffic." *40th IEEE/ACM International Symposium on Microarchitecture*, 2007.

LEE91 Lee, R., Kwok, A., and Briggs, F. "The Floating Point Performance of a Superscalar SPARC Processor." *Proceedings*, *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

LEE10 Lee, B., et al. "Phase-Change Technology and the Future of Main Memory." *IEEE Micro*, January/February 2010.

LEAN06 Lean, E., and Maccabe, A. "Reducing Memory Bandwidth for Chip-Multiprocessors using Cache Injection." *15th IEEE Symposium on High-Performance Interconnects*, August 2007.

LEON07 Leonard, T. "Dragged Kicking and Screaming: Source Multicore." *Proceedings*, *Game Developers Conference 2007*, March 2007.

LILJ88 Lilja, D. "Reducing the Branch Penalty in Pipelined Processors." *Computer*, July 1988.

LILJ93 Lilja, D. "Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons." *ACM Computing Surveys*, September 1993.

LILJ00 Lilja, D. *Measuring Computer Performance: A Practitioner's Guide.* Cambridge, UK: Cambridge University Press, 2000.

LITT61 Little, J. "A Proof for the Queuing Formula: $L = \lambda W$." *Operations Research*, May–June 1961.

LITT11 Little, J. "Little's Law as Viewed on its 50th Anniversary." *Operations Research*, May–June 2011.

LOVE96 Lovett, T., and Clapp, R. "Implementation and Performance of a CC-NUMA System." *Proceedings, 23rd Annual International Symposium on Computer Architecture*, May 1996.

LUND77 Lunde, A. "Empirical Evaluation of Some Features of Instruction Set Processor Architectures." *Communications of the ACM*, March 1977.

MACD84 MacDougall, M. "Instruction-level Program and Process Modeling." *IEEE Computer*, July 1984.

MANJ01a Manjikian, N. "More Enhancements of the SimpleScalar Tool Set." *Computer Architecture News*, September 2001.

MANJ01b Manjikian, N. "Multiprocessor Enhancements of the SimpleScalar Tool Set." *Computer Architecture News*, March 2001.

MASH04 Mashey, J. "War of the Benchmark Means: Time for a Truce." *Computer Architecture News*, September 2004.

MASH95 Mashey, J. "CISC vs. RISC (or what is RISC really)." *USENET comp.arch newsgroup, article 46782*, February 1995.

MAK97 Mak, P., et al. "Shared-Cache Clusters in a System with a Fully Shared Memory." *IBM Journal of Research and Development*, July/September 1997.

MAYB84 Mayberry, W., and Efland, G. "Cache Boosts Multiprocessor Performance." *Computer Design*, November 1984.

MCDO05 McDougall, R. "Extreme Software Scaling." *ACM Queue*, September 2005.

MCDO06 McDougall, R., and Laudon, J. "Multi-Core Microprocessors are Here." *; login*, October 2006.

MCMA93 McMahon, F., "L.L.N.L Fortran Kernels Test." *Source*, October 1993. www.netlib.org/benchmark/livermore

MOOR65 Moore, G. "Cramming More Components Onto Integrated Circuits." *Electronics Magazine*, April 19, 1965. Reprinted in *Proceedings of the IEEE*, January 1998.

MORR74 Morris, M. "Kiviat Graphs—Conventions and Figures of Merit." *ACM SIGMETRICS Performance Evaluation Review*, October 1974.

MORS78 Morse, S., Pohlman, W., and Ravenel, B. "The Intel 8086 Microprocessor: A 16-bit Evolution of the 8080." *Computer*, June 1978.

MYER78 Myers, G. "The Evaluation of Expressions in a Storage-to-Storage Architecture." *Computer Architecture News*, June 1978.

NASM12 The NASM Development Team. *NASM—The Netwide Assembler.* http://nasm.us/, 2012.

NOVI93 Novitsky, J., Azimi, M., and Ghaznavi, R. "Optimizing Systems Performance Based on Pentium Processors." *Proceedings, COMPCON '92*, February 1993.

NVID09 NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi." *NVIDIA White Paper*, August 2009.

NVID14 NVIDIA, "CUDA C Programming Guide." *NVIDIA Documentation*, 2014.

OWEN08 Owens, J., et al. "GPU Computing." *Proceedings of the IEEE*, May 2008.

PADE81 Padegs, A. "System/360 and Beyond." *IBM Journal of Research and Development*, September 1981.

PARH10 Parhami, B. *Computer Arithmetic: Algorithms and Hardware Design.* Oxford: Oxford University Press, 2010.

PATT82a Patterson, D., and Sequin, C. "A VLSI RISC." *Computer*, September 1982.

PATT82b Patterson, D., and Piepho, R. "Assessing RISCs in High-Level Language Support." *IEEE Micro*, November 1982.

PATT84 Patterson, D. "RISC Watch." *Computer Architecture News*, March 1984.

PATT85a Patterson, D. "Reduced Instruction Set Computers." *Communications of the ACM*. January 1985.

PATT85b Patterson, D., and Hennessy, J. "Response to 'Computers, Complexity, and Controversy.'" *Computer*, November 1985.

PATT88 Patterson, D., Gibson, G., and Katz, R. "A Case for Redundant Arrays of Inexpensive Disks (RAID)." *Proceedings*, *ACM SIGMOD Conference of Management of Data*, June 1988.

PEDD14 Peddle, J. "Inside Intel's Gen 8 GPU." *EE Times*, September 22, 2014.

PEIR99 Peir, J., Hsu, W., and Smith, A. "Functional Implementation Techniques for CPU Cache Memories." *IEEE Transactions on Computers*, February 1999.

PELE97 Peleg, A., Wilkie, S., and Weiser, U. "Intel MMX for Multimedia PCs." *Communications of the ACM*, January 1997.

PFIS98 Pfister, G. *In Search of Clusters.* Upper Saddle River, NJ: Prentice Hall, 1998.

PHAN07 Phanslkar, A., Joshi, A., and John, L. "Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite." *ACM International Symposium on Computer Architecture, ISCA'07*, 2007.

POLL99 Pollack, F. "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies" (keynote address). *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999.

PRES01 Pressel, D. "Fundamental Limitations on the Use of Prefetching and Stream Buffers for Scientific Applications." *Proceedings*, *ACM Symposium on Applied Computing*, March 2001.

PROP11 Prophet, G. "Use GPUs to Boost Acceleration." *IDN*, December 2, 2011.

PRZY88 Przybylski, S., Horowitz, M., and Hennessy, J. "Performance Trade-offs in Cache Design." *Proceedings*, *15th Annual International Symposium on Computer Architecture*, June 1988.

PRZY90 Przybylski, S. "The Performance Impact of Block Size and Fetch Strategies." *Proceedings*, *17th Annual International Symposium on Computer Architecture*, May 1990.

RADI83 Radin, G. "The 801 Minicomputer." *IBM Journal of Research and Development*, May 1983.

RAGA83 Ragan-Kelley, R., and Clark, R. "Applying RISC Theory to a Large Computer." *Computer Design*, November 1983.

RAOU09 Raouk, S., et al. "Phase-Change Random Access Memory: A Scalable Technology." *IBM Journal of Research and Development*, July/September 2008.

RECH98 Reches, S., and Weiss, S. "Implementation and Analysis of Path History in Dynamic Branch Prediction Schemes." *IEEE Transactions on Computers*, August 1998.

REDD76 Reddi, S., and Feustel, E. "A Conceptual Framework for Computer Architecture." *Computing Surveys*, June 1976.

REIM06 Reimer, J. "Valve Goes Multicore." *ars technica*, November 5, 2006. arstechnica.com/articles/paedia/cpu/valve-multicore.ars

ROBI07 Robin, P. "Experiment with Linux and ARM Thumb-2 ISA." *Embedded Linux Conference*, 2007.

RODR01 Rodriguez, M., Perez, J., and Pulido, J. "An Educational Tool for Testing Caches on Symmetric Multiprocessors." *Microprocessors and Microsystems*, June 2001.

SAND10 Sanders, J., and Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Reading, MA: Addison-Wesley Professional, 2010.

SATY81 Satyanarayanan, M., and Bhandarkar, D. "Design Trade-Offs in VAX-11 Translation Buffer Organization." *Computer*, December 1981.

SEBE76 Sebern, M. "A Minicomputer-compatible Microcomputer System: The DEC LSI-11." *Proceedings of the IEEE,* June 1976.

SERL86 Serlin, O. "MIPS, Dhrystones, and Other Tales." *Datamation*, June 1, 1986.

SHAN38 Shannon, C. "Symbolic Analysis of Relay and Switching Circuits." *AIEE Transactions*, Vol. 57, 1938.

SHAR03 Sharma, A. *Advanced Semiconductor Memories: Architectures*, *Designs*, *and Applications.* New York: IEEE Press, 2003.

SHUM13 Shum, C., Susaba, F., and Jacobi, C. "IBM zEC12: The Third-Generation High-Frequency Mainframe Microprocessor." *IEEE Micro*, March/April 2013.

SIEW82 Siewiorek, D., Bell, C., and Newell, A. *Computer Structures: Principles and Examples.* New York: McGraw-Hill, 1982.

SIMO96 Simon, H. *The Sciences of the Artificial.* Cambridge, MA: MIT Press, 1996.

SLAV12 Slavici, V., et al. "Adapting Irregular Computations to Large CPU-GPU Clusters in the MADNESS Framework." *IEEE International Conference on Cluster Computing*, 2012.

SMIT82 Smith, A. "Cache Memories." *ACM Computing Surveys*, September 1982.

SMIT87 Smith, A. "Line (Block) Size Choice for CPU Cache Memories." *IEEE Transactions on Communications,* September 1987.

SMIT88 Smith, J. "Characterizing Computer Performance with a Single Number." *Communications of the ACM*, October 1988.

SMIT89 Smith, M., Johnson, M., and Horowitz, M. "Limits on Multiple Instruction Issue." *Proceedings*, *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

SMIT95 Smith, J., and Sohi, G. "The Microarchitecture of Superscalar Processors." *Proceedings of the IEEE*, December 1995.

SOHI90 Sohi, G. "Instruction Issue Logic for High-Performance Interruptable, Multiple Functional Unit, Pipelined Computers." *IEEE Transactions on Computers*, March 1990.

STAL14a Stallings, W. "Gigabit Wi-Fi." *Internet Protocol Journal*, September 2014.

STAL14b Stallings, W. "Gigabit Ethernet." *Internet Protocol Journal*, December 2014.

STAL15 Stallings, W. *Operating Systems*, *Internals and Design Principles*, *Eighth Edition.* Upper Saddle River, NJ: Pearson, 2015.

STEN90 Stenstrom, P. "A Survey of Cache Coherence Schemes of Multiprocessors." *Computer*, June 1990.

STEV64 Stevens, W. "The Structure of System/360, Part II: System Implementation." *IBM Systems Journal*, Vol. 3, No. 2, 1964. Reprinted in [SIEW82].

STEV13 Stevens, A. *Introduction to AMBS 4 ACE and big.Little Processing Technology.* ARM White Paper, July 29, 2013.

STRE78 Strecker, W. "VAX-11/780: A Virtual Address Extension to the DEC PDP-11 Family." *Proceedings*, *National Computer Conference*, 1978.

STRE83 Strecker, W. "Transient Behavior of Cache Memories." *ACM Transactions on Computer Systems*, November 1983.

STRI79 Stritter, E., and Gunter, T. "A Microprocessor Architecture for a Changing World: The Motorola 68000." *Computer*, February 1979.

TAMI83 Tamir, Y., and Sequin, C. "Strategies for Managing the Register File in RISC." *IEEE Transactions on Computers*, November 1983.

TANE78 Tanenbaum, A. "Implications of Structured Programming for Machine Architecture." *Communications of the ACM*, March 1978.

TI12 Texas Instruments. *66AK2H12/06 Multicore DSP+ARM KeyStone II System-on-Chip (SoC).* Data Manual SPRS866, November 2012.

TJAD70 Tjaden, G., and Flynn, M. "Detection and Parallel Execution of Independent Instructions." *IEEE Transactions on Computers,* October 1970.

TOON81 Toong, H., and Gupta, A. "An Architectural Comparison of Contemporary 16-Bit Microprocessors." *IEEE Micro*, May 1981.

TUCK67 Tucker, S. "Microprogram Control for System/360." *IBM Systems Journal,* No. 4, 1967.

UNGE02 Ungerer, T., Rubic, B., and Silc, J. "Multithreaded Processors." *The Computer Journal*, No. 3, 2002.

UNGE03 Ungerer, T., Rubic, B., and Silc, J. "A Survey of Processors with Explicit Multithreading." *ACM Computing Surveys*, March 2003.

VANC14 Vance, A. "99% of the World's Mobile Devices Contain an ARM Chip." *Business Week*, February 10, 2014.

VONN45 Von Neumann, J. *First Draft of a Report on the EDVAC.* Moore School, University of Pennsylvania, 1945. Reprinted in *IEEE Annals on the History of Computing*, No. 4, 1993.

VRAN80 Vranesic, Z., and Thurber, K. "Teaching Computer Structures." *Computer*, June 1980.

WALL85 Wallich, P. "Toward Simpler, Faster Computers." *IEEE Spectrum*, August 1985.

WANG99 Wang, G., and Tafti, D. "Performance Enhancement on Microprocessors with Hierarchical Memory Systems for Solving Large Sparse Linear Systems." *International Journal of Supercomputing Applications*, Vol. 13, 1999.

WEIC90 Weicker, R. "An Overview of Common Benchmarks." *Computer*, December 1990.

WEIN75 Weinberg, G. *An Introduction to General Systems Thinking.* New York: Wiley, 1975.

WEIS84 Weiss, S., and Smith, J. "Instruction Issue Logic in Pipelined Supercomputers." *IEEE Transactions on Computers,* November 1984.

WHIT97 Whitney, S., et al. "The SGI Origin Software Environment and Application Performance." *Proceedings*, *COMPCON Spring '97*, February 1997.

WILK51 Wilkes, M. "The Best Way to Design an Automatic Calculating Machine." *Proceedings, Manchester University Computer Inaugural Conference,* July 1951.

WILK53 Wilkes, M., and Stringer, J. "Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer." *Proceedings of the Cambridge Philosophical Society*, April 1953. Reprinted in [SIEW82].

WILL90 Williams, F., and Steven, G. "Address and Data Register Separation on the M68000 Family." *Computer Architecture News*, June 1990.

YEH91 Yeh, T., and Patt, N. "Two-Level Adapting Training Branch Prediction." *Proceedings*, *24th Annual International Symposium on Microarchitecture*, 1991.

ZHOU09 Zhou, P., et al. "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology." *ACM International Symposium on Computer Architecture, ISCA'09*, 2009.

INDEX

A

Absolute address, 483
Absolute scalability, 633
Access control, 313–314
Access time (latency), 123
Accumulator (AC), 14, 85, 418
ACE (Advanced Extensible Interface Coherence Extensions), 674–675
Acorn RISC Machine (ARM), 34. *See also* ARM architecture
Active secondary clustering method, 635*t*
Adders, 392–396
  4-bit, 394
  implementation of an, 395
  multiple-bit, 394–395
  single-bit, 394
Addition, 337–340
  binary, 392
  overflow rule, 337–338
  twos complement, 338–339
Addressable units, 122
Address bus, 101
Address generation sequencing, 743–744
Addressing modes, 457–463
  ARM, 466–469, 526–527
  autoindexing, 462
  base-register, 462
  basic, 459
  direct, 459
  displacement, 461–463
  effective address (EA), 458
  immediate, 459
  indexing, 462–463
  indirect, 459–460
  Intel x86, 463–466
  MIPS R4000, 560–561
  mode field, 458
  PC-relative, 461
  postindexing, 462–463
  preindexing, 463
  register, 460–461
  register indirect, 461
  relative, 461
  SPARC, 568
  stack, 463
Address lines, 101
Address modify instructions, 17
Address registers, 492
Address space, 304–305
Advanced RISC Machines. *See* ARM architecture
Alignment check (AC), 519
Alignment Mask (AM), 521
Allocation, Pentium 4 processor, 517
Amdahl, Gene, 53
Amdahl's law, 53–55, 660
American Standard Code for Information Interchange (ASCII), 232, 421, 422
AND gate, 389
AND operation, 430
Antidependency, 509, 586
Application-level parallelism, 662
Application processors, 31–32
Application programming interface (API), 42
Arithmetic and logic unit (ALU), 490, 494, 542
  addition, 337–340
  ARM Cortex-A8, 600–601
  division, 347–350
  flag values, 329–330
  floating-point notation, 350–358
  IAS computer, 11, 13, 16
  IBM system/360, 22
  IBM 3033 microinstruction execution, 755
  inputs and outputs, 330
  integers, 330–350
  multicore computer, 8
  multiplication, 340–347
  operands for, 329
  single-processor computer, 6
  SPARC architecture, 567
  subtraction, 337–340
  Texas Instruments 8800 Software Development Board (SDB), 762–765
Arithmetic instructions, 416, 421, 429
Arithmetic mean, 60, 62
Arithmetic operations, 429, 431
Arithmetic shift, 345, 431
ARM addressing modes, 466–469, 526–527
  abort mode, 527
  branch instruction, 468
  data processing instructions, 468
  exception modes, 526, 527
  fast interrupt mode, 527
  indexing methods, 466–467
  interrupt mode, 527
  load and store, 466–468
  load/store multiple addressing, 468–469
  offset value, 466–467
  postindexing, 468
  preindexing, 467
  privileged modes, 526
  supervisor mode, 527

ARM addressing modes (*continued*)
system mode, 527
undefined mode, 527
user mode, 526
ARM architecture, 33–39
ACE cache line states, 675
alignment checking, 424
branch instructions, 444
data-processing instructions, 444
data types, 423–425
Endian support, 425
evolution, 34
extend instructions, 444
instruction set, 34–35
instructions of, 417
load and store instructions, 444
multiply instructions, 444
nonaligned access, 423
parallel addition and subtraction instructions, 444
products, 35–39
SETEND instruction, 425
status register access instructions, 444
unaligned access, 424
use of condition codes in, 444–445
VLSI, 34
ARM Cortex-A8, 596–604
address generation unit (AGU), 599
branch target buffer (BTB), 599
dual-issue restrictions, 602
flow of instructions, block diagram, 597
global history buffer (GHB), 599
in-order issue, 597–598
instruction decode unit, 599–600
instruction fetch unit, 598–599
integer execute unit, 600–603
integer pipeline, 598
load/store pipeline, 602
memory system effects on instruction timings, 601
SIMD and floating-point instructions, 603–604
ARM Cortex-A15 core, 670–673
ACE cache line states, 675
energy consumption, 673
pipelines, 672
ARM Cortex-A15 MPCore, 677–681
active interrupt, 679
block diagram of, 677–678
cache coherency, 680–681
core, 677
CPU interface, 680
debug unit and interface, 677
direct data intervention (DDI), 681
duplicated tag RAMs, 681
generic interrupt controller (GIC), 677
generic timer, 677
hardware interrupts, 680
inactive interrupt, 679
interprocessor interrupts (IPIs), 679
interrupt handling, 678–680
L1 cache coherency, 681
L2 cache coherency, 681
legacy FIQ line, 680
migratory lines, 681
pending interrupt, 679
private timer and/or watchdog interrupts, 680
program trace, 677
snoop control unit (SCU), 677, 680–681
ARM Cortex-A7 core, 671–673
ACE cache line states, 675
energy consumption, 673
pipelines, 672
ARM Cortex-M3, 604–607
branch forwarding, 606
branch speculation, 606
bus matrix, 605
data watchpoint and trace (DWT), 605
dealing with branches, 606–607
debug access port, 605
decode stage, 606
embedded trace macrocell, 605
flash patch and breakpoint unit, 604
memory protection unit, 604
nested vectored interrupt controller (NVIC), 604
pipeline, 607
pipeline structure, 605–606
processor core, 604
serial wire viewer, 605
Thumb-2 instruction, 605–606
wake-up interrupt controller (NVIC), 604
ARM instruction format, 479–482
immediate constants, 479
Thumb instruction set, 479–481
Thumb-2 instruction set, 481–482
ARM memory management, 309–314
access control, 313–314
Domain Access Control Register, 314
formats, 310–313
organization of memory, 309–310
parameters, 313
translation lookaside buffer (TLB), 309–310
virtual memory address translation, 310–311
ARM processor, 524–530
attributes, 525
interrupt processing, 529–530
processor organization, 525
registers, 527–529
Array processor, 615
Assemblers, 761, 763
Assembly language, 413, 415, 482–484. *See also* Instruction formats
BASIC statement, 482

pseudoinstruction, 483
symbolic program in, 483
Asserting, signal, 377
Associative access, 123
Associative mapping, 138–140
Associative memory, 123
Autoindexing, 462
Auxiliary memory, 127

B

Backward compatibility, 29
Balanced transmission, 105
Bank groups, 184
Base, 307
Base address, 297
Base digit, 319
Base-register addressing, 462
Batch system, 280
Bell Labs, 17
Benchmark programs, 68
BFU (binary floating-point unit), 10
Biased representation, 351
Big endian ordering, 452
Big.Little Chip, 671
Binary adder, 339
Binary addition, 392
Binary Coded Decimal (BCD), 384
Binary system, 321
Bit-interleaved parity disk performance (RAID level 3), 210–211
Bit length conversion, 332
Bit ordering, endian, 455
Blade servers, 638–639
Blocked multithreaded scalar, 631
Blocked multithreaded superscalar, 632
Blocked multithreaded VLIW, 632
Blocked multithreading, 630
Block-level distributed parity disk performance (RAID level 5), 212
Block-level parity disk performance (RAID level 4), 211–212
Block multiplexor, 262
Blocks, 122, 690
Booth's algorithm, 346–347
cache, 160
I/O, 408
logic, 408
m, 129, 134–135
memory, 133, 137, 140–142, 619
packets or protocol, 257
process control, 494
SDRAMs, 182
SPLD, 406
tape, 222
thread, 690–691, 696
Blu-ray DVD, 217, 221
Boole, George, 373
Boolean algebra, 373–375, 392
AND operation, 374
basic identities of, 375
Boolean operators, 375
exclusive-or (XOR) operation, 374
NAND function, 374
NOT operation, 374
OR operation, 374
Boolean functions, implementation of
algebraic simplification, 381
canonical form, 381
Karnaugh maps, 381–386
NAND and NOR gates, 388
Quine–McCluskey method, 384–387
rules for simplification, 382–383
sum of products (SOP) form, 379, 380
of three combinations, 379
Boolean (logic) instructions, 416
Booth's algorithm, 344–347
Branches
conditional instructions, 509–515
control hazard (branch hazard), 508–509
as correlator, 515
Cortex-M3 processor, 606–607
delayed, 515, 557–558
dynamic strategies, 51–512
history approach, 513–515
history table, 513
instruction fetch stage, 513–514
loop buffer for, 510–511
loop-closing, 515
microinstructions, 744
multiple streams for, 509–510
pipelining and, 509–515
prediction, 511–515, 589, 593–594
prefetched branch target, 510
Branch prediction, 47–48
Branch target buffer (BTB), 511, 513, 593
British Broadcasting Corporation (BBC), 34
Buffers, 83
Bus arbitration technique, I/O, 243
Bus interconnection, 100–102
Bus master, 146
Bus watching approach, 146
Bus width, 25–27
Byte, 111
Byte multiplexor, 262
Byte ordering, endian, 452

C

Cache, 6
banking, 703
Cortex-R, 35
injection, 259
miss, 130, 146, 152, 258–259, 261, 310, 581, 594, 630, 632, 677, 681

Cache coherence, 621–624
directory protocols, 623
hardware-based solutions, 623–624
multicore computers, 674–675
snoopy protocols, 623–624
software, 622–623
write-invalidate approach, 624
write policies, 622
write-update protocol, 624
Cache-coherent nonuniform memory access (CC-NUMA), 640
advantages and disadvantages, 643
organization, 641–642
Cache Disable (CD), 521
Cache hit, 130, 148
Cache line, 133
Cache memory, 128–149, 536
addresses, 131–133
high-performance computing (HPC), 131
lines, 129
line size, 129, 147
logical cache, 132
mapping function, 133–144
multilevel cache, 147–149
number of caches, 147–149
physical cache, 132–133
read operation, 129–130
replacement algorithms, 145
sizes, 133, 134
split cache, 149
structure of, 128, 129
tag, 129
unified cache, 149
virtual address, 133
virtual cache, 132
write policy, 145–147
Cache miss, 130
Cache set, 140
Call/return instructions, 438–439
Capacitors, 20
Carry lookahead, 395
CD-ROM, 217
CD-RW, 217
Central processing unit (CPU), 83, 689
of general-purpose microcomputer, 25
Intel 8085, 721
interconnection, 6
internal structure, 490
involvement in I/O channels, 261–262
memory and, 83
multicore computer, 6, 667–671
performance and performance per watt, 692
in second generation computers, 18
single-processor computer, 4
vs. GPU, 691–692
with system bus, 490
Chaining, 301
Character data operands, 470
Characteristic table, 397
Chip multiprocessing, 630
Chip multiprocessor (multicore), 628–633, 657
Chips, 7–8, 21
ARM, 34
control store, 752
DDR, 183
DRAM memory, 172–173
EPROM package of, 172–173
four-core, 52
high-speed, 50
integrated circuit, 21, 24
Intel Quad-Core Xeon processor, 8–9
I/O controller, 8
LSI, 751
memory, 8, 9, 25, 47–48, 172–173
microcontroller, 32
microprocessor, 32
multicore, 102, 268, 657, 663, 665, 668, 682
PU, 683
RAM, 390
semiconductor memory, 170–172
two-core, 52
ultra-large-scale integration (ULSI), 24
Chipset, PCI Express, 108
Clock (bus) cycle, 57
Clocked S–R flip-flop, 397–399
Clock rate, 57
Clock speed, 57
Clock tick, 57
Cloud auditor, 648–649
Cloud broker, 648–649
Cloud carrier, 648–649
Cloud computing, 39–42
actors, 648–649
broad network access, 644
community cloud, 646
computing, 39
deployment models, 646
elements, 643–647
essential characteristics of, 644–645
hybrid cloud, 646
infrastructure as a service (IaaS), 42
measured service, 644
networking, 40
on-demand self-service, 644–645
platform as a service (PaaS), 41
private cloud, 646
public cloud, 646
rapid elasticity, 644
reference architecture, 647–649
resource pooling, 645
service models (SaaS, PaaS, IaaS), 645–646, 649

software as a service (SaaS), 40–41
storage, 40
Cloud consumers, 649
Cloud provider, 648–649
Clusters, 615, 633–639
active secondary, 635
benefits and limitations, 633, 635
blade servers, 638–639
compared to SMP, 639
computer architecture, 637–638
configurations, 633–636
load balancing, 636
middleware services and functions, 638
operating system design, 636–637
parallelizing computation, 636–637
passive standby, 635
separate server, 635
server, 634
shared-disk, 634
shared disk approach, 636
shared nothing approach, 636
single-system image, 637
Coarse-grained threading, 663
Combinational circuit
Boolean equations, 378–388
decoders, 390–392
defined, 378
graphical symbols, 378
multiplexers, 388–390
read-only memory (ROM), 392
sequential circuits, 396–405
truth table of, 378
Commercial computers, 11
Committing (retiring) instructions, 590
Communication devices, 231
Community cloud, 646
Compact disk (CD), 217–220
CD recordable (CD-R), 219–220
CD rewritable (CD-RW), 220
CD-ROM (compact disk read-only memory), 217–219
CD-R optical disk, 220
Compaction, I/O memory, 296
Compiler-based register optimization, 547–549
Complex instruction set computer (CISC), 27, 540
characteristics, 538
motivations for contemporary, 549–551
vs. RISC design, 553–555, 570–571
Complex PLDs (CPLDs), 406, 408
Computer architecture, 2
Computer instruction, 413
Computer on a chip, 32
Computer organization, 2
Computers
architecture, 2
components, 81–83
family concept, 536
function, 83–98
fundamental elements of, 11
generations, 17
generation-to-generation compatibility of, 2
history of, 11–27
instruction, fetch and execute function, 84–89
instruction set architecture (ISA), 2
interconnection structures, 99–100
organization, 2
structure and function, 3–11
Computer system performance
Amdahl's law, 53–55
benchmark principles, 67–68
calculating a mean value, 59–67
clock speed, 57
designing for, 46–52
following improvements in chip organization and architecture, 50–52
general-purpose computing on GPUs (GPGPU), 52–53
graphics processing units (GPUs), 52–53
instruction execution rate, 58–59
Little's law, 55–56
many integrated core (MIC), 52
microprocessor speed, 47–48
multiple processors, 52
performance balance, 48–49
SPEC benchmarks, 68–74
Conditional branch instructions, 433
Conditional jump, 439
Condition codes, 492, 744, 757
advantages and disadvantages, 493
ARM architecture, 444–445
EFLAGS register, 518
Intel x86, 438–440
program status word (PSW), 495
RISC-based machines, 560
Constant angular velocity (CAV), 198, 218
Constant linear velocity (CLV), 198, 218
Control, 85
access, 313–314
interrupt, 720
I/O modules, 231, 232–233
lines, 101–102
logical, 13
machine instructions, 415
storage control (SC), 683
and timing, 232–233
Control hazard (branch hazard), pipelining, 508–509
Control hazards, pipelining, 509

Controllers
cache, 146, 623
disk, 107
disk drive, 231
fanouts, 269
I/O, 108, 121, 235, 236, 262
mass storage, 35
memory and peripheral, 657, 668
microcontrollers, 32, 187
network interface, 107
Control lines, 101
Control registers, 519–521
Control signals, 716–719
Control unit (CU), 4, 6, 490
characterization of, 715
control signals, 716–719
execute cycle, 712–713
fetch cycle, 709–711
functional requirements, 714–715
hardwired implementation, 724–727
IAS computer, 11, 13
indirect cycle, 711–712
inputs and outputs, 716–717
instruction cycle, 713–714
internal processor organization and, 719–720
interrupt cycle, 712
micro-operations, 708–714
of processor, 714–724
COP (dedicated co-processor), 11
Core i7 EE 4960X microprocessor, 29
Cortex-A and Cortex-A50, 35
Cortex-M series processors, 35–39
analog interfaces, 38
bus matrix, 38
clock management, 38
core, 38
debug access port (DAP), 36
debug logic, 36
embedded trace macrocell (ETM) module, 36
energy management, 38
ICode interface, 38
memory, 38
memory protection unit, 38
nested vector interrupt controller (NVIC), 36
parallel I/O ports, 38
peripheral bus, 39
security, 38
serial interfaces, 38
SRAM & peripheral interface, 38
32-bit bus, 39
timers and triggers, 38
Cortex-R, 35
Counters, 402–405
ripple, 402–403
synchronous, 403–404
Texas Instruments 8800 Software Development Board (SDB), 759
C programming, 159
CRAY C90, 122
CUDA (Compute Unified Device Architecture), 689–691
cores, 690, 696, 697
CUDA core/SM count, 694
programming language, 689, 690
Current program status registers (CPSR), ARM, 527
Cycles per instruction (CPI) for a program, 58
Cycle stealing, 249
Cycle time, 57–58, 525, 562, 620
instruction, 18, 501, 503, 716
memory, 18, 58, 123
pipeline, 504–506
processor, 58
Cyclic redundancy check (CRC), 106

D

Daisy chain technique, I/O, 243
Database scaling, 618
Data buffering, I/O modules, 233
Data bus, 101
Data cache, 152
Data channel, 18
Data communication, 4
Data exchanges, 636
Data flow, instruction cycles, 497–499
Data flow analysis, 48
Data formatting, magnetic disks, 196–199
Data hazards, pipelining, 508–509
Data (bus) lines, 101
Data-L2, 11
Data movement, 4
Data processing, 4, 20, 85, 416, 421, 444, 601, 667
ARM, 525
instruction addressing, 468
load/store model of, 525
machine instructions, 415
Data processing instruction addressing, 468
Data registers, 491
Data storage, 4, 20, 40, 42, 124, 167, 265, 416
machine instructions, 415
Data transfer, 427–428
IAS computer, 16
instructions, 427–428
I/O modules, 231
packetized, 103
Data types
ARM architecture, 423–425
IEEE 754 standard, 424
Intel x86 architecture, 422–423
packed SIMD, 422
Debug access port (DAP), 36

Debug logic, 36
Decimal system, 319–320
Decode instruction unit, Cortex-A8 processor, 599
Decoders, 390–392, 595
  as demultiplexer, 390, 392
DEC P DP- 8, 23–24
Dedicated processor, 32
Deeply embedded systems, 32–33
Delayed branch, pipelining, 557–558
Delayed load, pipelining, 558–559
Delay slot, 557
Demand paging, 299–300
DeMorgan's theorem, 375, 377, 388
Device communication, I/O modules, 233
D flip-flop, 399, 401
DFU (decimal floating-point unit), 10
Differential signaling, 105
Digital computer, 20
Digital Equipment Corporation (DEC), PDP series computers, 23
Digital logic
  Boolean algebra, 373–375
  combinational circuits, 378–396
  gates, 376–378
  programmable logic device (PLD), 405–409
  sequential circuits, 396–405
Digital versatile disk (DVD), 217, 220–221
Direct access, 122–123
Direct-access device, 223
Direct address, 463
Direct addressing, 459, 473
Direct cache access (DCA), 254–261
  performance issue and benefits, 257–259
  strategies, 259
Direct Data I/O, 254
  cache write operation, 260
  comparison of DMA with, 260
  packet input, 259–261
  packet output, 259–261
  strategy, 261
  TCP/IP protocol handler, 261
  write-back strategy, 260
  write-through strategy, 260
Direction flag (DF), 519
Directives, 704
Direct mapping technique, 134–138
Direct memory access (DMA), 98
  comparison of DDIO with, 260
  control/command registers of Intel 8237, 253–254
  8237 DMA usage of system bus, 252
  fly-by DMA controller, 253
  function, 249–251
  interrupt breakpoints during an instruction cycle, 250
  programmed and interrupt-driven I/O, 248–249
  SMPs, 619
  using shared last-level cache, 255–257
Directory protocols, 623
Dirty (use) bit, 146
Disabled interrupt, 95
Discrete components, 20–21
Disk cache memory, 158
Disk drive, I/O, 232
Dispatcher, 288
Displacement addressing, 461–463
  mode, 465
Distributed arbitration, 118
Dividend, 347
Division, 347–350
  flowchart for unsigned binary, 348
  partial remainder, 347–349
  twos complement, 349
Divisor, 347
Double data rate, 183
Double-data-rate DRAM (DDR RAM), 183–184
Double-sided disks, 200
Drive, Pentium 4 processor, 517
Dual redundancy disk performance (RAID level 6), 212
DVD, 217
DVD-R, 217
DVD-ROM, 217
DVD-RW, 217
Dynamic access random memory (DRAM) technology, 104
Dynamic RAM (DRAM), 147, 167–168, 171–172, 187
  DDR SDRAM, 183–184
  pin configuration, 172
  synchronous DRAM (SDRAM), 181–182

E

EAS/390 memory system, 432
Edge-triggered flip-flop, 403
EDVAC (Electronic Discrete Variable Computer), 11
Effective address (EA), 458, 460
EFLAGS register, Intel x86 processors, 518–519
Electrically erasable programmable read-only memory (EEPROM), 170
Embedded Microprocessor Benchmark Consortium (EEMBC) benchmark, 482
Embedded systems, 29–33
  deeply, 32–33
  operating system (OS), 31
  organization, 29–30
Embedded trace macrocell (ETM) module, 36
Emulation (EM), 520

Enabled interrupt, 95, 712
Encoded microinstruction format, 748–751
Erasable programmable read-only memory (EPROM), 170, 172
Error control function, 106
Error-correcting codes, 175
Error correction, 216–217
  semiconductor memory, 174–180
Error detection, I/O modules, 234
ESCON (Enterprise Systems Connection), 269
Ethernet, 265–266
Exceptions, interrupts and, 522–523, 529
Excitation table, 403
Execute cycle, 84, 87, 92
  micro-operations (micro-ops), 712–713
Execution. *See also* Program execution
  fetch and instruction, 496–497
  fetched instruction, 85
  IBM 3033 microinstruction, 743, 754–755
  instruction execution rate, 58–59
  I/O program, 89, 91
  of loads and stores in MIPS R4000 microprocessor, 565
  LSI-11 microinstruction, 751–754
  microprogramming, 745–755
  multithreading, 628
  RISC instruction, 537–542
  speculative, 48
  superscalar, 48, 589–590
Expansion boards, 7
Exponent overflow, 358
Exponent value, 351
Extended Binary Coded Decimal Interchange Code (EBCDIC), 421, 432
Extension Type (ET), 520
External interface standards, 263–266
External memory, 39, 121–122, 127, 185, 187
  magnetic disk, 195–203
  magnetic tape, 222–224
  optical-disk systems, 217–222
  RAID, 204–213
  solid state drives (SSDs), 212–216

F

Failback, 636
Failover, 636
Failure management, clusters, 636
Family concept, 536
Fanouts, 269
Fetch cycle, 15, 84, 85, 87, 92, 93, 469, 497–498, 709–711
  micro-operations (micro-ops), 709–711
Fetched code bits, 175
Fetch instruction unit, 489, 496
  Cortex-A8 processor, 599
  execution of, 85
Fetch overlap, pipelining, 501
Field-programmable gate array (FPGA), 406–409
  I/O blocks, 408
  logic block, 409
  structure, 408
Fine-grained threading, 663
FireWire Serial Bus, 264
Firmware, 107, 215, 731
First generation of computers. *See* IAS computer
First-in first-out (FIFO) algorithm, 145
Fixed-head disk, 199
Fixed-point representation, 335
Fixed-size partitions, 294–295
Flag, register organization, 527–528
Flags. *See* Condition codes
Flash memory, 170, 185–187
  NOR and NAND, 186–187
  operation, 185–186
Flip-flops, 396–400
  basic, 400
  clocked S–R, 397–399
  D, 399, 401
  edge-triggered, 403
  J–K, 399–400, 402–403
Flit, 104
Floating-point arithmetic, 358–367, 424, 576, 583, 697
  addition, 359–361
  division, 361
  exponent overflow, 358
  exponent underflow, 358
  IEEE standard for binary, 365–367
  minus infinity, 365
  multiplication, 361–362
  normalization, 361
  precision considerations, 362–365
  rounding to plus, 365
  round to nearest, 364
  round toward zero, 365
  significand overflow, 359
  significand underflow, 359
  subtraction, 359–361
Floating-point notation, 350–358
  base, 351
  biased representation, 351
  with binary numbers, 350–352
  exponent value, 351
  of IBM S/390 architecture, 353–354
  IEEE standard for binary, 354–358
  negative overflow, 353
  negative underflow, 353
  normal number, 351–352
  positive overflow, 353
  positive underflow, 353

principles, 350–354
significand, 351–352
sign of, 351
Floating-point operations per second (FLOPs), 692
Floppy disk, 200
Floppy (contact) magnetic disks, 196, 199
Flow control function, 104, 106, 109, 115
Flow dependency, 580
FORTRAN programs, 159, 282, 540
Fractions, 322–324
Frames, I/O memory, 269
Front end, Pentium 4 processor, 593–594
Fully nested interrupt mode, 243
Functional encoding, 750
Functional mean, 60
Functions, 276–280
I/O, 98, 232–234
FXU (fixed-point unit), 10

G
Gaps, magnetic disks, 197
Gates, 20, 376–378
delay, 376
functionally complete sets of, 377
NAND, 377
NOR, 377–378
GeForce 8800 GTX, 693
General-purpose computing using a GPU (GPGPU), 52–53, 689
General purpose register, 460–462, 466, 491–492, 517–518, 528
Geometric mean, 60, 64–67
Gigabit Ethernet, 107
Global history buffer (GHB), 599
Gradual underflow, 367
Graphical symbol, 376, 378
Graphics processing units (GPUs), 52–53, 689
architecture overview, 692–701
as a coprocessor, 704–706
CUDA cores, 696–697
dual warp scheduler, 696–697
Fermi, 694
floating-point operations per second for, 693
floating-point (FP) unit pipeline, 697
GDDR5 (graphic double data rate), 694–695
of Gen8 architecture, 701–704
grid and block dimensions, 691
hardware components equivalence mapping, 691
integer (INT) unit pipeline, 697
load and store units, 697
L1 cache, 697–700
memory hierarchy attributes, 698
memory types, 700–701
multicore computers, 667–669
NVIDIA, 693–694
performance and performance per watt, 692
processor cores, 690
read-after-write (RAW) data hazard, 701
registers, 697–700
shared memory, 697–700
special function units (SFU), 694, 697
streaming multiprocessor architecture, 695–700
streaming multiprocessors (SMs), 691
vs. CPU, 691–692
Graphics technology interface (GTI), 704
Guard bits, 362

H
Hamming, Richard, 176
Hamming code, 176
Hard disk, 199, 201
Hard disk drives (HDDs), 212, 214
parameters, 201
Hard failure, 174
Hardware cache coherence, 623–624
Hardware transparency approach, 146
Hardwired implementation, 724–727
control unit inputs, 725–726
control unit logic, 726–727
Hardwired programs, 82
Harmonic mean, 60, 62–64
Hash functions, 300
Hashing technique, 301
Heterogeneous System Architecture (HSA) Foundation, 669
Hexadecimal, 324–326
Hexadecimal digits, 325
Hexadecimal notation, 324–326
High-definition optical disks (HD DVD), 221–222
High-performance computing (HPC), 131
Hit ratio, 138, 140, 144
Horizontal loss, 632
Host channel adapter (HCA), 269
Human-readable devices, 230
Hybrid cloud, 646
Hybrid threading, 663

I
IAS computer, 11
arithmetic and logic unit (ALU), 11, 14, 16
computation of addresses, 17
conditional branch instruction, 17
control unit, 12, 14
data transfer, 16
execute cycle, 16
fetch cycle, 15
flowchart of, 15
input–output (I/O) equipment, 12

IAS computer (*continued*)
instruction cycle, 15
instruction groups, 16–17
logical control, 13
memory of, 11, 14–15
operation code (opcode) instruction, 14, 16
registers, 14–15
storage locations, 14
structure of, 12
unconditional branch instruction, 16
von Neumann's earlier proposal, 12–14
IA-64 architecture, 492
IBM 801 system, 558
IBM 7094, 18, 19
configuration, 18
Instruction Backup Register, 18
IBM system/360, 22–23
ALU, 23
CPU, 23
third generation of computers, 22–23
IBM 370/168, pipeline streams of, 510
IBM 360/91, pipeline streams of, 510
IBM 3033, pipeline streams of, 510
IBM 3033 microinstruction execution, 743, 754–755
IBM zEnterprise EC12 I/O
channel path, 268
channels, 268
channel structure, 266–268
channel subsystems (CSS), 267
hardware system area (HSA), 267
I/O frames–front view, 269
I/O system organization, 268–270
logical partition, 267
subchannel, 268
system assist processor (SAP), 267
Z frame, 268
IBM zEnterprise EC12 mainframe computer, 9
cache structure, 683–684
embedded DRAM (eDRAM), 684
multichip module (MCM), 682
organization, 682–683
processor node structure, 682
processor unit (PU), 683
storage control (SC), 683
I-cache, 11
ICode interface, 38
Identification flag (ID), 519
IDU (instruction decode unit), 10
If-Then (IT) instruction, 481
IFU (instruction fetch unit), 9
Immediate address, 459
Immediate addressing mode, 459
Immediate constants, ARM, 479–480
Incremental scalability, 633
Indexed address, 492
Indexing, 462–463
Index registers, 462–463, 492
Indirect addressing, 459–460
Indirect cycle, 711–712
Indirect instruction cycle, 458
InfiniBand, 263, 265, 269
Infinity, IEEE interpretation, 365
Infinity arithmetic, 365
Information technology (IT), 31
Infrastructure as a service (IaaS), 42, 646
In-order completion, 583
In-order issue, 583–585
Input–output (I/O) process, 4–5
Institute of Electrical and Electronics Engineers (IEEE) standards
for binary floating-point arithmetic, 365–367
double-precision floating-point numbers, 560
802.11 Wi-Fi, 266–267
802.3, 265
802.3 for ethernet, 265
floating-point representations, 422
1394 for FireWire, 264
for rounding, 364
754 Subnormal Numbers, 366–367
754-1985 floating-point arithmetic standard, 697
Instr-L2, 11
Instruction address register, 87–88
Instruction buffer register (IBR), 14
Instruction cache, Pentium 4, 150
Instruction cycle, 84, 85, 87, 496–499, 713–714
data operation (do), 88
execute cycle, 496, 498
fetch and instruction execution activities, 496–497
fetch cycle, 496–498
instruction address calculation (iac), 87–88
instruction fetch (if), 88
instruction operation decoding (iod), 88
interrupts and, 91–96
interrupt stage, 496
operand address calculation (oac), 88
operand fetch (of), 88
operand store (os), 88
Instruction cycle code (ICC), 713
Instruction execution rate, 58–59
Instruction formats. *See also* Assembly language
ADD instruction, 557
addressing bits, 470–471
allocation of bits, 470–473
ARM, 479–482
DEC-10 instructions, 540
granularity of addressing, 471
high-level language (HLL), 537, 539–542, 545
If-Then (IT) instruction, 481
Intel x86, 477–479

JUMP instruction, 557
length, 469–470
memory-transfer length, 469, 470
MIPS R4000 microprocessor, 560–561
multiple instructions per cycle, 632
NOOP instruction, 557
operand address, 470
Patterson programs, 539
PDP-8, 471–472
PDP-11, 474
PDP-10, 472–473
range of addresses, 471
reduced instruction set architectures, 551–553
register vs. memory address, 470–471
SETHI instruction, 569
set of registers, 471
SPARC (Scalable Processor Architecture), 568–569
S/390 Move Characters (MVC) instruction, 553
32-bit Thumb instructions, 482
variable-length instructions, 473–477
VAX, 474–477, 479, 537, 539, 540
Instruction issue, 582–583
Instruction pipelining, 500–516, 536, 576, 596, 628, 672
branch prediction, 511–515
control hazard (branch hazard), 508–509
data hazard, 508–509
dealing with conditional branches, 509–515, 516
delayed branch, 515, 557–558
delayed load, 558
Intel x86 architecture, 593
loop buffer, 510–511
loop unrolling, 559
measures of pipeline performance, 504–507
MIPS R4000 microprocessor, 561–565
multiple streams, 509–510
optimization of, 557–559
pipeline bubble, 507
prefetched branch target, 510
reduced instruction set computer (RISC), 555–559
with regular instructions, 555–556
resource hazard, 507–508
strategy, 500–504
Instruction prefetch (fetch overlap), 501
Instruction register (IR), 14, 85, 493, 497, 710
Instruction set architecture (ISA), 2, 58, 278, 482, 667, 671
ARM, 35, 481
Thumb-2, 35, 481
Instruction sets. *See* Addressing modes
Instruction window, 585
Integers, 321–322
addition, 337–340
division, 347–350
fixed-point, 335
negation, 336–337
nonnegative, 330, 424
overflow rule, 337–338
packed byte and packed byte, 422
packed doubleword and packed doubleword, 423
packed quadword and packed qaudword, 423
packed single-precision floating-point and packed double- precision floating-point, 423
packed word and packed word, 422
radix point, 330
range extension, 333–335
signed, 424, 568
sign magnitude, 331
subtraction, 337–340
twos complement operation of, 331–333, 336, 342–347
unsigned, 330
Integrated circuit (IC), 7, 20–22
pattern, 21
Integrated memory controller (IMC), 255
Intel cache evolution, 150
Intel Core i7-990X, 676–677
Intel Core microarchitecture, 592
Intel 82C55A programmable peripheral interface, 245–248
Intel 82C59A interrupt controller, 243–244
Interrupt Acknowledge (INTA) of, 243
Interrupt Request (INTR) of, 243
responsibility of, 243
Intel 3420 chipset, 9
Intel 8085, 720–724
Address Latch Enabled (ALE) pulse signals, 723
control unit, 721
CPU block diagram, 721
external signals, 722
incrementer/decrementer address latch, 720
interrupt control, 720
machine cycles, 721
OUT instruction, 723–724
pin configuration, 723
serial I/O control, 720
Intel 80386
interrupt modes of, 243–244
multiple I/O modules of, 243
user-visible register organization for, 496
Intel 80486 pipelining
condition codes in, 516
decoding, 515
execute cycle, 515
fetch cycle, 515
write back stage, 516

Intel 8237A
DMA controller, 251–253
DMA registers, 254
Intel 8255A programmable peripheral interface
architecture and operation, 245–247
keyboard/display terminal, 247–248
operating modes and configurations, 246–247
pin layout, 245–246
Intel Gen8 architecture, 701–704
execution unit (EU), 701
memory modules, 703
registers, 701
SIMD floating-point units, 701
simultaneous multithreading (SMT) architecture, 701
subslices, 702
Intel HD Graphics 5300 Gen8, 704
Intel microprocessors
Core M Processor, 704
present times, 27
1970s, 26
1980s, 26
1990s, 26
Intel Quad-Core Xeon processor chips, 8–9
Intel x86 addressing modes, 463–466
based scaled index with displacement mode, 466
base mode, 465
base with displacement mode, 465–466
base with index and displacement mode, 466
displacement mode, 465
immediate mode, 464
mode calculation, 464
register operand mode, 464
relative addressing, 466
scaled index with displacement mode, 466
segment registers for, 464
Intel x86 architecture, 27–29
allocate stage, 595
branch prediction strategy, 593–594
branch target buffer (BTB), 593–594
cache/memory parameters, 592
control registers, 519–521
Core series microprocessor, 28
data hazards, 508
data types, 422–423
decode unit, 594–595
dispatching, 596
8080 microprocessor, 28
8086 microprocessor, 28
80286 microprocessor, 28
80386 microprocessor, 28
80486 microprocessor, 28
evolution of, 28
front ends, 593–595
hardware registers, 595
instruction fetch unit, 594
instruction queue unit, 594–595
instruction set, 28
instruction translation lookaside buffer (ITLB), 594
integer and floating-point register files, 596
interrupt processing, 522–524
microarchitecture, 591–596
micro-op queuing, 596
micro-op scheduling, 596
out-of-order execution logic, 595–596
Pentium series microprocessor, 28
pipelining, 593
predecode unit, 594
register organization, 517–522
register renaming, 595–596
reorder buffer (ROB) entry, 595
static prediction algorithm, 594
Intel x86 instruction format, 477–479
address size, 478
displacement field, 478
instruction prefixes, 478
ModR/M byte, 478
opcode field, 478
operand size, 478
segment override, 478
SIB byte, 478
Intel x86 memory management, 304–309
address spaces, 304–305
4-Gbyte linear memory space, 308
logical address in, 305
OS design and implementation, 305
parameters, 307
privilege level and access attribute, 305
requested privilege level (RPL), 306
segmented paged memory, 305
segmented unpaged memory, 304
segment number, 306
table indicator (TI), 305
unsegmented paged memory, 304
unsegmented unpaged memory, 304
virtual memory in, 305
Intel x86 operation types
call/return mechanism, 438–439
memory management, 439
MMX instructions, 440–442
SIMD instructions, 440–444
status flags and condition codes, 439–440
Intel x86 processor family
exception and interrupt vector table, 524
exceptions, 522–523
interrupt-handling routine, 523–524
interrupt processing, 522–524
register organization, 517–522
Intel Xeon processors
direct cache access strategies, 259

E5-2600/4600, 255–257
multicore processors, 255
Interconnection structures, 99–100
bus interconnection, 100–102
point-to-point, 100, 102–107
Interconnection transfers
I/O to or from memory, 100
I/O to processor, 100
memory to processor, 100
processor to I/O, 100
processor to memory, 100
Interleaved memory, 173–174
Interleaved multithreaded scalar, 630–631
Interleaved multithreading, 629–630
Interleaved multithreading superscalar, 632
Interleaved multithreading VLIW, 632
Intermediate queues, 293
Internal processor bus, 490–491
International Reference Alphabet (IRA), 232, 421
Internet of things (IoT), 30–31
Interrecord gaps, 222
Interrupt-driven I/O, 239–248
bus arbitration, 243
daisy-chain configuration, 243
design issues, 241–243
Intel 82C59A interrupt controller, 243–244
interrupt-handler program, 241
interrupt processing, 239–241
multiple interrupt lines, 242
software poll, 242
TESTI/O, 242
vectored interrupt, 243
Interrupt enable flag (IF), 518
Interrupts, 89–98
classes, 89
control lines, 102
cycle, 92, 498, 712
disabled, 95
handler, 91, 93
instruction cycle and, 91–96
multiple, 95–98
nested interrupt processing, 97
point of view of user program, 92
program flow of control without and with, 90
sequential interrupt processing, 97
Interrupt service routine (ISR), 95
Intertrack gaps, 197
Inverted page table, 301
I/O address register (I/OAR), 83
I/O buffer (I/OBR) register, 83
I/O channels, 261–263
architecture, 263
block multiplexor, 262
byte multiplexor, 262
characteristics of, 262
function, 261–262
IBM zEnterprise EC12 I/O, 268–270
multiplexor channel, 262
selector channel, 262
I/O command, 235
I/O components, 83
I/O controllers, 108, 121, 235, 236, 262
I/O devices
disk drive, 232
external, 230–232
human-readable devices, 230
keyboard/monitor arrangement, 231–232
machine-readable devices, 230
I/O driver software, 214
I/O functions, 98
I/O hub (IOH), 103–104
I/O-memory transfer, 98
I/O modules, 715
address recognition, 233
command decoding, 233
control function, 231, 232–233
control lines, 101–102
data buffering, 233
data transfer, 231
device communication, 231, 233
error detection, 234
field-programmable logic array, 408
functions or requirements for, 232–234
interconnection structures, 100
interconnection transfers, 100
interface to, 231
I/O requests in RAID schemes, 211
machine instructions, 414
PCIe TLP transaction types, 113
processor communication, 233
QPI connections, 103–104
read-write operation, 145–146
registers and, 494
semiconductor memory, 173
status reporting, 233
structure, 234–235
timing, 203, 232–233
transducer, role of, 231
I/O privilege flag (IOPL), 519
I/O processor, 235
I/O program
execution of, 89, 91
time required for, 94
IPC (instructions per cycle), 628
Isolated I/O, 237
ISU (instruction sequence unit), 9

J

Java applications, 662
Java Virtual Machine, 662
JEDEC Solid State Technology Association, 183

J–K flip-flop, 399–400, 402–403
Job control language (JCL), 282
Job program, 280–282
Jump instruction, 433

K
Karnaugh maps, 381–386
Kernel (nucleus), 279
Khronos Group's OpenCL, 689
K-way set associative cache organization, 140–142

L
Lands, compact disks, 218
Lane, 104
Large-scale integration (LSI), 24
Last-in-first-out (LIFO) queue, 463
L1 cache, 128
L3 cache, 128
L2 cache, 128
L2 control, 11
Least-frequently used (LFU) algorithm, 132, 145
Least-recently used (LRU) algorithm, 132, 145, 299
Least significant digit, 319
Linear tape-open (LTO) system, 224
Linear tape-open (LTO) tape drives, 224
Linking, 40, 281, 474
Link layer, 105–107, 115
Links, InfiniBand, 265
Linux, 18
Little endian ordering, 455
Little's law, 55–56
Load balancing, clusters, 636
Load/store addressing, ARM, 466–468
Load/store multiple addressing, ARM, 468–469
Locality of reference, 125, 128, 158
Local variable, 437
Locked operation, 113
Logical address, 297
Logical cache, 132
Logical data operands, 421–422
Logical operations (opcode), 429
Logical shift, 430
Logic block, 406, 408
Logic (Boolean) instructions, 417
Logic-memory performance balance, 48–50
Long-term data storage function, 4
Long-term scheduling, 287–288
Lookup table, 408
Loop buffer, pipelining, 510–511
Loop unrolling, pipelining, 559
Low-voltage differential signaling (LVDS), 105
LSI-11 microinstruction execution, 751–754
LSU (load-store unit), 10

M
Machine cycles, 721
Machine instructions. *See also* Instruction cycle; Instruction formats
  addresses, 417–419
  arithmetic instructions, 416
  ARM architecture, 417
  BASIC instruction, 416
  branch instructions, 433–434
  conditional branch instruction, 433
  conversion instructions, 432
  data transfer instructions, 427–428
  elements of, 413–414
  high-level language, 416
  increment-and-skip-if-zero(ISZ) instruction, 434
  input/output instructions, 432
  instruction register (IR), 415
  instruction set design, 419
  I/O instructions, 416
  logic (Boolean) instructions, 416
  memory instructions, 416
  MMX instructions, 440–442
  multiple-address instructions, 419
  next instruction reference, 414
  operands, 420–422
  operations (opcode), 413
  reduced instruction set computer (RISC), 419
  result operand reference, 414
  SETEND instruction, 425
  skip instructions, 434
  source and result operands, 414
  source operand reference, 413
  stacks and, 418
  symbolic representation, 415
  system control instructions, 432
  test instructions, 416
  transfer-of-control instructions, 433–438
  unconditional branch instruction, 434
  zero-address instructions, 418
Machine parallelism, 581–582, 588–589
Machine-readable devices, 230
Magnetic-core memory, 24
Magnetic disk
  access time, 201
  contemporary rigid, 196
  cylinder, 200
  data organization and formatting, 196–199
  double-sided disks, 200
  intertrack gaps, 197
  multiple platters, 200
  multiple zone recording (MZR), 198
  performance parameters, 201–203
  physical characteristics, 199–201
  read and write mechanisms, 195–196

rotational delay, 201–202
rotational positional sensing (RPS), 202
sectors, 197
seek time, 201, 202
sequential organization, 203
single-sided disks, 200
timing, 203
transfer time, 202
Magnetic RAM (MRAM), 189
Magnetic tapes, 222–224
Magnetic tunneling junction (MTJ), 189
Magnetoresistive (MR) sensor, 196
Mainframe computers, 23, 47
Mantissa, 351
Many integrated core (MIC), 52
Mapping function of cache memory, 133–144
associative, 138–140
direct, 134–138
k-way set associative cache organization, 140–142
set-associative, 140–144
Mask, 430
Medium-term scheduling, 288
Memory, 83. *See also* Cache memory
access time (latency), 123
associative access, 123
auxiliary, 127
bank, 173
capacity, 121
characteristics, 121–127
concepts for internal, 121–122
control lines, 102
cycle time, 123
direct access, 122–123
hierarchy, 124–127
instructions, 416
interconnection structures, 99–100
interconnection transfers, 100
levels of, 125
locality of reference, 125
method of accessing units, 122–123
"natural" unit of organization of, 122
noncacheable, 146
PCIe TLP transaction types, 113
performance parameters, 123
physical types of, 123
random access, 123
read-only memory (ROM), 124
real, 300
secondary, 127
sequential access, 122
transfer rate, 123
two-level, 157–164
Memory address register (MAR), 14, 83, 493–494, 497, 709–710
Memory bank, 173
Memory buffer register (MBR), 14, 83, 493–494, 497, 499, 709–710, 712–713
Memory cell, 20
Memory controller hub (MCH), 255–256
Memory cycle time, 18, 123
Memory hierarchy, 124–127
Memory instructions, 416
Memory management
ARM, 309–314
base addresses, 297
compaction, 296
Intel x86, 304–309
intermediate queue, 293
logical addresses, 297
page frames, 297
page table, 298
paging, 297, 308–309
partitioning, 294–297
physical addresses, 297
segmentation, 303–306
SMP, 621
swapping, 293–294
time-consuming procedure, 296
translation lookaside buffer (TLB), 301–303
virtual memory, 299–301
Memory management unit (MMU), 35, 132, 310, 458
Cortex-A and Cortex-A50, 35
Cortex-R, 35
Memory-mapped I/O, 237–238
Memory modules, 83, 84, 99
Memory protection, OS, 289
Memory Protection Unit (MPU), 35
MESI (modified/exclusive/shared/invalid) protocol, 621–627
line states, 625
L1-L2 cache consistency, 627
read hit, 626
read miss, 626
read-with-intent-to-modify (RWITM), 626
state transition diagram, 625
write hit, 627
write miss, 626–627
Metallization, 20
Microcomputers, 3, 24
Microcontroller chip, 32–33
Microelectronics, 19–23
control unit, 20
data movement, 20
data processing, 20
data storage, 20
development of, 20–22
Microinstruction bus (MIB), 751
Microinstruction spectrum, 747

Micro-operations (micro-ops), 152, 708–714
execute cycle, 712–713
fetch cycle, 709–711
indirect cycle, 711–712
instruction cycle, 713–714
instruction set, 715
interrupt cycle, 712
rules, 711
sequencing, 715
time units, 711
Microprocessor chips, 32
Microprocessor register organizations, 495–496
Microprocessors, 25–26
Microprogrammed control units, 536, 733–735
Microprogrammed implementation, 6
Microprogramming, 727, 730
address generation techniques, 743–744
advantages, 737
design considerations, 739–740
disadvantages, 737
encoding, 748–751
execution, 745–755
hard, 748
horizontal, 748
interrupt testing, 744
LSI-11 microinstruction execution, 751–754
LSI-11 microinstruction sequencing, 744–745
microinstructions, 730–733
microprogrammed control unit, 733–735
next sequential address, 744
opcode mapping, 744
sequencing techniques, 740–742
soft, 748
subroutine facility, 744
taxonomy, 745–748
vertical, 748
Wilkes control, 735–739
Microprogramming language, 731
Migratory lines, 681
Millions of floating-point operations per second (MFLOPS) rate, 59
Millions of instructions per second (MIPS) rate, 59
Minuend, 338
MIPS rate, 59
MIPS R4000 microprocessor, 559–565
enhancing pipelining, 563
execution of loads and stores, 565
instruction set, 560–561
partitioning of chip, 560
pipelining instructions, 561–565
Miss, 126, 138
MMX (multimedia task)
instructions, 440–442, 444, 521
registers, 521–522
Mnemonics, 415, 761
Monitor (simple batch OS), 281
Monitor arrangement, I/O, 231
Monitor Coprocessor (MP), 520
Moore, Gordon, 21
Moore's law, 21, 47, 51, 692
consequences of, 21–22
Most significant digit, 319
Motherboard, 7
Motorola MC68000 microprocessor registers, 495
Movable-head disk, 199
Multicore computers, 6–8
arithmetic and logic unit (ALU), 8
cache coherence, 674–675
cache memory, 6
central processing unit (CPU), 6, 667–671
cores, 6, 8
digital signal processors (DSPs), 669–671
equivalent instruction set architectures, 671–674
external memory interface (EMIF), 671
graphics processing units (GPUs), 667–669
hardware performance, 657–660
heterogeneous multicore organization, 667–675
homogenous multicore organization, 667
instruction logic, 8
levels of cache, 665–666
load/store logic, 8
memory subsystem memory controller (MSMC), 675
MOESI model, 675
motherboard, 7–8
multicore shared memory (MSM), 671
multicore shared memory controller (MSMC), 671
organization, 665–667
power consumption, 659–660
printed circuit board (PCB), 7
processor, 6, 8
simplified view of major elements of, 7
software performance, 660–665
Multicore processors, 6, 8, 657
Multilane distribution, 105
Multilevel cache memory, 147–149
Multiple-bit adders, 394–395
Multiple instruction, multiple data (MIMD) stream, 615
Multiple instruction, single data (MISD) stream, 615
Multiple interrupt lines, I/O, 242
Multiple parallel processing, 628
Multiple platters, magnetic disks, 200
Multiple streams, pipelining, 509–510
Multiplexers, 388–390
in digital circuits to control signal and data routing, 389

4-to-1, 388–389
using AND, OR, and NOT gates, 389
Multiplexor, 18
Multiplexor channel, 262
Multiple zone recording (MZR), 198, 218
Multiplicand, 340–341
Multiplication
arithmetic shift, 345
Booth's algorithm, 344–347
flowchart for unsigned binary, 342
hardware implementation of unsigned binary, 341
twos complement, 342–347
unsigned integers, 330
Multiplier quotient (MQ), 14
Multiprocessor OS design, SMP considerations for, 284
Multithreading, 628–633
blocked, 630–632
coarse-grained, 630
fine-grained, 629
implicit and explicit, 628–629
interleaved, 629–632
principal approaches, 629–630
process, 628–629
process switch, 629
resource ownership, 628
scheduling/execution, 628
simultaneous (SMT), 630, 632–633, 667
thread, 629
thread switch, 629

N

NAND flash memory, 186–187, 188, 214
NAND gate, 377, 388
NaNs, IEEE standards, 365–366
N-disk array, 212
Negation, integers, 336–337
Negative overflow, 353
Negative underflow, 353
Nested Task (NT) flag, 519
Nested vector interrupt controller (NVIC), 36
Neumann, John von, 11, 81
Nibble, 324
NIST SP-800-145, 39
NIST SP 500-292 *(NIST Cloud Computing Reference Architecture)*, 647–648
Noncacheable memory approach, 146
Nonredundant disk performance (RAID level 0), 205
Nonremovable disk, 199
Nonuniform memory access (NUMA) machines, 614, 615, 640–643
advantages and disadvantages, 643
motivation, 640–641
organization, 641–642
processor 3 on node 2 (P2-3) requests, 642
Nonvolatile memory, 124, 127
Nonvolatile RAM technologies, 188, 190
NOR flash memory, 186–188
NOR gate, 377
Normalized numbers, 67
NOR S–R latch, 398
NOT operation, 429
Not Write Through (NW), 521
Number system
base digit, 319
binary system, 321
converting between binary and decimal, 321–324
decimal system, 319–320
fractions, 322–324
hexadecimal notation, 324–326
integers, 321–322
least significant digit, 319
most significant digit, 319
nibble, 324
positional number system, 320
radix point, 320
Numeric Error (NE), 521
NVIDIA's CUDA, 689

O

Offset addressing, ARM, 467
Omnibus, 24
Operands, 420–422
bit-oriented view, 422
characters, 421
logical data, 421–422
numbers, 420–421
packed decimal, 420–421
Operating system (OS), 494
batch system, 280
functions, 276–280
interactive, 280
interrupt-driven I/O or DMA operations, 285–286
interrupts, 283
memory management, 286
memory protection, 283
Multics OS, 287
multiprogramming batch, 283–286
objectives, 276–280
privileged instructions, 283
scheduling, 280, 287–293
setup time, 280
simple batch, 281–283
symmetric multiprocessors (SMPs), 617–621
timer, 283
time-sharing, 286–287
types of, 280–287
uniprogramming, 286

Operational technology (OT), 31
Operations (opcode), 425–438
  AND, 430
  arithmetic shift operation, 431
  ARM architecture, 444–445
  basic arithmetic, 429
  common instruction set, 426–427
  conversion, 432
  data transfer, 427–428
  IBM EAS/390 data transfer operations, 428
  input/output, 432
  Intel x86 operation types, 438–444
  logical, 429–431
  nested procedures, 435, 436
  NOT, 429
  procedure call instructions, 435–438
  process actions for various, 427
  reentrant procedure, 436
  rotate or cyclic shift, 430–431
  stack frame, 437
  system control, 432
  transfer-of-control, 433–438
  XOR, 430
Optical memory, 195
  characteristics, 222
  compact disk (CD), 217–220
  high-definition optical disks, 221
OR gate, 376
Original equipment manufacturers (OEMs), 24
Orthogonality, 472–473
Out-of-order execution, 595–596
Out-of-order issue, 585–586
Output dependency, 509, 579, 583
Overflow, 337

P

Packed decimal representation, 421
Packets, data, 109
Page fault, 299
Page frame, 297
Page-level cache disable (PCD), 521
Page-level writes transparent (PWT) bit controls, 521
Page replacement, 300
Pages, 297
Page tables, 298, 300–301
Paging, 297–298, 303–304, 521
  demand, 299–300
  virtual memory, 158
  x86, 308–309, 463
Parallelism, 576
  application-level, 662
  fundamental limitations to, 579–581
  instruction-level, 579, 581–582
  machine-level, 581–582, 588–589
  multicore computers, 657–659
  procedural dependency and, 581
  process-level, 662
  resource conflict and, 581
  thread-level, 662
  true data dependency and, 579–581
Parallelized application, 637
Parallelizing compiler, 637
Parallel organizations, 615–617
Parallel processing
  cache coherence, 621–624
  chip multiprocessing, 630
  cloud computing, 643–649
  clusters, 633–639
  MESI (modified/exclusive/shared/invalid) protocol, 624–627
  multiple instruction, multiple data (MIMD) stream, 615, 617
  multiple instruction, single data (MISD) stream, 615
  multiple processor organizations, 615–617
  multithreading, 628–633
  nonuniform memory access (NUMA), 640–643
  single instruction, multiple data (SIMD) stream, 615, 617
  single instruction, single data (SISD) stream, 615
  symmetric multiprocessors (SMP), 617–621
  write policies, 622
Parallel recording, 222
Parallel register, 401
Parameters, magnetic disks, 201–203
Parametric computing, 637
Parity bits, 176
Partial product, 341
Partial remainder, 347–349
Partitioning, I/O memory management, 294–297
Pascal, 159
Passive standby clustering method, 635
Patterson programs, 539
PCI Express (PCIe), 104, 107–115, 214, 265, 704
  address spaces and transaction types, 113–114
  data link layer packets, 115
  devices that implement, 108–109
  I/O device or controller, 108
  I/O drawers, 270
  legacy endpoint category, 109
  multilane distribution, 110
  ordered set block, 111
  physical layer, 109–111
  protocol architecture, 109
  root complex, 108
  TLP packet assembly, 114–115
  transaction layer (TL), 112–115
  transaction layer packet processing, 115
  Type 0 and Type 1 configuration cycles, 114

PCI Special Interest Group (SIG), 107
PC-relative addressing, 461
PDP-8 Bus Structure, main memory, 24
PDP-8 instruction format, 471–472
PDP-8 instruction format design, 471–472
PDP-11 instruction format design, 474
PDP-11 processor, 87
PDP-10 instruction format, 472–473
PDP-10 instruction format design, 472–473
Pentium 4 cache, 149–152
  execution unit, 152
  fetch/decode unit, 150
  instruction cache, 152
  memory subsystem, 152
  operating modes, 152
  out-of-order execution logic, 150
  write-back policy, 152
Peripheral component interconnect (PCI), 107
Peripheral (external) devices, I/O, 233
Personal technology, 31
Phase change disk, 220
Phase-change RAM (PCRAM), 188, 189
  SET and RESET operation, 189
Phit (physical unit), 104
Physical address, 297
Physical cache, 133
Physical characteristics of data storage, 124
Physical layer, 104–105
Physical records, 222
Physical types of memory, 123
Pipelining, 47. *See also* Instruction pipelining
Pit, 218
Platform as a service (PaaS), 41, 646
Platters, 195, 200
Point-to-point interconnection structures, 24, 100, 102–107
  QPI link layer, 105–107
  QPI physical layer, 104–105
  QPI protocol layer, 107
  QPI routing layer, 107
Point-to-point interfaces, 255
Polarization-current-induced magnetization switching, 189
Pollack's rule, 660
Polycarbonates, 217
POP stack operation, 469, 474
Positional number system, 320
Positive overflow, 353
Positive underflow, 353
Postindexing, 462–463
Power consumption, 659–660
Power density, 50
Preindexing, 463
Printed circuit board (PCB), 7
Printers, 230
Private cloud, 646
Privileged instructions, 283
Procedural dependency, parallelism, 581
Procedure return, 438
Process
  block, 107, 489, 560
  control block, 289
  multithreading, 628–633
  resource ownership, 628
  scheduling, 287–293
  states, 288–290
  switch, 629
Process-level parallelism, 662
Processor organization, 489–491
  common fields or flags, 494
  functional elements of, 715
  requirements, 489
Processors
  interconnection structures, 100
  interconnection transfers, 100
  1970s, 26
  1980s, 26
  1990s, 26
  present times, 27
Product of sums (POS), 380
Program counter (PC), 14, 84, 289, 389–390, 493, 497, 499, 710
Program execution
  example, 86
  execute cycle, 84
  fetch cycle, 84, 87
  fetched instruction, execution of, 85
  instruction cycle, 84, 85, 87
  interrupts, 89–98
  I/O program, 89, 91
Programmable array logic (PAL), 406
Programmable logic array (PLA), 405–406
Programmable logic device (PLD), 405–409
  complex PLDs (CPLDs), 408
  field-programmable logic array, 406–409
  programmable logic array (PLA), 405–406
  simple PLD (SPLD), 406
  terminology, 406
Programmable read-only memory (PROM), 169, 170
Programmed I/O
  commands, 236, 238
  instructions, 236–238
  interrupt-driven I/O and, 239–248
  isolated, 237
  memory-mapped, 237
  memory-mapped I/O, 237–238
  overview of, 236
  techniques, 235

Programming, 83
Program status word (PSW), 494
Protection Enable (PE), 520
Pseudoinstruction, 483
Public cloud, 646
Pushdown list, 463

Q

Queues, 55
  I/O operations, 267
QuickPath Interconnect (QPI), 102–107
  balanced transmission, 105
  differential signaling, 105
  direct connections, 103
  error control function, 106
  flow control function, 106
  layered protocol architecture, 103
  multiple direct connections, 103
  packetized data transfer, 103
  physical Interface, 105
  QPI link layer, 105–107
  QPI physical layer, 104–105
  QPI protocol layer, 107
  QPI routing layer, 107
  use on multicore computer, 103
Quiet NaN, 365–366
Quine-McCluskey method, 384–388

R

Radix point, 320, 330
RAID (Redundant Array of Independent Disks), 195, 204–213
  comparison, 213
  RAID level 5, 212
  RAID level 4, 211–212
  RAID level 1, 209–210
  RAID level 6, 212
  RAID level 3, 210–211
  RAID level 2, 210
  RAID level 0, 205–209
Random access, 123
Random-access memory (RAM), 167
Rate metric measures, 71, 73
Read hit/miss, 626
Read mechanisms, magnetic disks, 196
Read-mostly memory, 170
Read-only memory (ROM), 124, 169–170, 392
  truth table for, 393
Read-with-intent-to-modify (RWITM), 626
Read-write dependency, 509
Real memory, 300
Recordable (CD-R), 219
Reduced instruction set computer (RISC), 3, 27, 536
  architecture, 549–555
  Berkeley study, 541–542, 565
  cache, 545–546
  characteristics, 538
  classic, 553–555
  compiler-based register optimization, 547–549
  complex instruction sets, 537
  conditional statements, 539
  elements of design, 537
  global variables, 545
  high-level language (HLL) and, 537, 539–542, 545
  instruction execution, 537–542
  large register file, 545–546
  line of reasoning of, 538
  one machine instruction per machine cycle, 551
  operands, 540–541
  operations, 539–540
  pipelining, 555–559
  procedure calls, 541
  qualitative assessment, 570–571
  quantitative assessment, 570–571
  referencing a local scalar, 546–547
  register to register operations, 551–552
  register windows, 543–545
  simple addressing modes, 552
  simple instruction formats, 552
  vs. CISC design, 553–555, 570–571
  window-based register file, 546–547
Redundant disk performance via Hamming code (RAID level 2), 210
Reentrant procedure, 436
Register addressing, 460–461, 551–552
Register file, instruction pipe line, 542–547
Register indirect addressing, 461
Register organization, 491–496
Register renaming, 586–587
Registers, 401–402, 490
  address, 492
  ARM, 527–529
  control and status, 491, 493–495, 518, 519–521
  in control of I/O operations, 494
  current program status register (CPSR), 527–529
  data, 491
  devoted to floating-point unit, 518
  EFLAGS and RFLAGS, 518–519
  general purpose, 491–492, 517–518, 528
  graphics processor unit (GPU), 697–700
  index, 492
  instruction register (IR), 493
  instruction set design, 419
  Intel x86, 517–524
  memory address register (MAR), 493–494, 497
  memory buffer register (MBR), 493–494, 497, 499

microprocessor register organizations, 495–496
MMX, 521–522
numeric, 518
program status, 527, 528
reduced instruction set computer (RISC), 543–545, 551–552
saved program status register (SPSR), 527–529
segment, 518
16-bit data, 496
software-visible, 527
tags, 518
Texas Instruments 8800 Software Development Board (SDB), 759
user-visible, 491–493, 496
Register-to-register organization, 551
Register window, 543–545
Relative address, 298
Relative addressing, 461
Removable disk, 199
Replacement algorithms, cache memory, 145
Resident monitor, 281
Resistive-capacitive (RC) delay, 50
Resistive RAM (ReRAM), 188, 189
Resource conflict, parallelism, 581
Resource encoding, 750
Resource hazard (structural hazard), pipelining, 507–508
Resource ownership process, 628
Resume flag (RF), 519
Retire, 598–600
Ripple counters, 402–403
Root complex, 108
Rotate (cyclic shift) operation, 431
Rotating interrupt mode, 244
Rotational delay (latency), magnetic disks, 201
Rotational positional sensing (RPS), 202
Rounding, 364–365
Rounding, IEEE standards, 364
RU (recovery unit), 10

S

Saturation arithmetic, 441
Scalar values, 452
Scheduling, 287–293
I/O queue, 292
long-term, 287–288
long-term queue, 292
medium-term, 288
process control block, 289–290
process states, 288–290
queuing diagram representation of processor, 292
short-term, 288–293
short-term queue, 292
techniques, 290–293
time-sharing system, 288
Secondary (auxiliary) memory, 127
Second generation computers, 17–18
CPU, 18
data channel, 18
multiplexor schedules, 18
Sectors, magnetic disks, 197
Seek time, magnetic disks, 202
Segmentation, Pentium II processor, 303–304
Segment pointers, 492
Selector channel, 262
Semantic gap, 537
Semiconductor memory, 24–25, 167, 174
address lines, 171
arrangement of cells in array, 170
chip logic, 170–172
chip packaging, 172–173
dynamic RAM (DRAM), 167–168
electrically erasable programmable read-only memory (EEPROM), 170
erasable programmable read-only memory (EPROM), 170
error correction in, 174–180
flash memory, 170
interleaved memory, 173–174
I/O module, 173
organization, 166
programmable ROM (PROM), 169, 170
random-access memory (RAM), 167
read-mostly memory, 170
read-only memory (ROM), 169–170
SRAM vs. DRAM, 169
static RAM (SRAM), 168–169
trade-offs among speed, density, and cost, 170
types, 167
write enable (WE) and output enable (OE) pins, 172, 173
Semiconductors, 127, 185, 214
Sensor/actuator technology, 31
Sequencing, 739–745
Sequential access, 122
Sequential-access device, 223
Sequential circuits, 396–405
counters, 402–405
flip-flops, 396–400
registers, 401–402
Sequential organization, magnetic disks, 203
Serial ATA (SATA) sockets, 9
Serial ATA (Serial Advanced Technology Attachment), 265
Serial recording, 222
Serpentine recording, 222
Server clustering approaches, 635
Set-associative mapping, 140–144

Setup time, operating system (OS) efficiency, 280–281
Shannon, Claude, 373
Shift registers, 401–402
Short-term data storage function, 4
Short-term scheduling, 288–290
Signaling NaN, 365–366
Signal lines, PCI, 99
Sign bit, 331
Sign extension, 334
Significand, 359
  overflow, 359
  underflow, 359
Sign-magnitude representation, 331
Silicon, 20
Simple PLD (SPLD), 406
Simultaneous multithreading (SMT), 630, 667
Single-error-correcting, double-error-detecting (SEC-DED) code, 179–180
Single error-correcting (SEC) code, 179
Single instruction, multiple data (SIMD) stream, 615
Single instruction, single data (SISD) stream, 615
Single large expensive disk (SLEP), 204
Single-processor computer, 4–6
  arithmetic and logic unit (ALU), 6
  central processing unit (CPU), 4
  internal structure of, 9
  main memory, 4
  processor, 4
  registers, 6
  system bus, 5
  system interconnection, 5
Single-sided disks, 200
Single-system image, 637
Single-threaded scalar, 630–631
Skip instructions, 434
Small Computer System Interface (SCSI), 264
Small-scale integration (SSI), 21, 405
Snoop control unit (SCU), 677
Snoopy protocols, cache coherence, 623–624
Soft errors, 175
Software, 18, 83
  cache coherence solutions, 622–623
  dynamic voltage and frequency scaling (DVFS), 673–674
  I/O driver, 214
  multicore computer performance, 660–665
  poll, 242
  processing models, 673–674
  software as a service (SaaS) model, 40–41
  Valve game threading, 663–665
Software as a service (SaaS), 40–41, 645
Software poll technique, I/O, 242
Solid-state component, 17, 20
Solid state drives (SSDs), 17, 187, 212–216
  compared to HDD, 214
  organization, 214–216
  practical issues, 216
SPARC (Scalable Processor Architecture), 542
  addressing modes, 568
  ALU operations, 567
  branch instruction, 568–569
  current window pointer (CWP), 566
  effective address (EA) of an operand, 568
  instruction format, 568–570
  instruction set, 567–568
  processor status register (PSR), 566
  register set, 565–567
  register window layout, 566
  Sun SPARC, 452
  UltraSPARC, 71, 300
  window invalid mask (WIM), 566
Spatial locality, 159, 160
SPEC documentation
  base metric, 71
  benchmark, 71
  peak metric, 71
  rate metric, 71
  reference machine, 71
  speed metric, 71
  system under test, 71
Special interest group (SIG), PCI, 107
Special mask interrupt mode, 244
Speculative execution, 48
Speed metric measures, 71
Speedup factor, 506–507
Speedup of the system, 53–55, 660–661
Spin-transfer torque RAM (STT-RAM), 188, 189
Split cache, 149
  memory, 147
S–R Latch, 396–398
Stack addressing, 463
Stack pointer, 492
Standard Performance Evaluation Corporation (SPEC) benchmarks
  floating-point benchmarks, 70
  integer benchmarks, 69, 72
  SPEC CPU2006, 68–71
  SPECint_base2006, 72
  SPECint_rate_base2006, 72
  SPECint_rate2006, 72
  SPECint2006, 72
  SPECjbb2013 (Java Business Benchmark), 68
  SPECjvm2008, 68
  SPECsfs2008, 68
  SPECviewperf, 68
  SPECvirt_sc2013, 68
  SPECwpc, 68
State diagrams, instruction cycles, 414
State of a process, 288–290
Static RAM (SRAM), 36, 38, 148, 168–169, 187

Status flags, 439
Status registers, 493–495
Status signals, I/O, 231–232
Stored-program concept, 11
Streaming multiprocessors (SMs), 691
Stripe, 205, 211, 212
Striped data, 211
Striped disk performance (RAID level 0), 205–209
Structured programming (SAL), 159
Subnormal numbers, 366–367
Substrate, 195
Subtraction, 337–340
  rule, 338
  twos complement, 338–339
Subtrahend, 338
Sum of products (SOP), 379
Superpipelined approach, 578–579
Superpipelined processor, 578–579
Superscalar, 9, 28, 51, 149, 474, 632
  branch prediction, 589
  committing or retiring of instruction, 590
  dependency in, 579–581
  execution, 48
  execution of programs, 589–590
  implementation, 590
  in-order completion, 583
  instruction fetch stage, 589
  instruction issue policy, 582–586
  instruction-level parallelism in, 581–582
  machine parallelism in, 581–582, 588–589
  organization, 577
  out-of-order completion, 583–586
  overview, 576–581
  pipelining and scheduling techniques, 152, 507
  processors, characteristics, 538
  register renaming, 586–587
  reported speedups, 577
  types of orderings, 582
  vs. superpipelining, 578–579
SuperSpeed, 264
Swapping, I/O memory management, 293–294
Switch, 108
Symmetric multiprocessors (SMPs), 614, 615, 617–621
  availability, 618
  bus organization, 620
  characteristics, 617
  DMA transfers, 619
  existence of multiple processors, 618
  incremental growth, 618
  memory and I/O channels, 619
  memory management, 621
  operating system of, 617–621
  performance, 617
  reliability and fault tolerance, 621
  scaling, 618
  scheduling, 621
  simultaneous concurrent processes, 621
  synchronization, 621
SYNCH byte, 199
Synchronous counter, 403–405
Synchronous DRAM (SDRAM), 181–182
  DDR SDRAM, 183–184
Syndrome words, 176
System buses, 5, 101
System control operations, 432
System interconnection (bus), 5
System Performance Evaluation Corporation (SPEC), 68. *See also* SPEC documentation
System software, 17

T

Tags, cache memory, 140
Task Switched (TS), 520
Temporal locality, 159–160
Test instructions, 416
Texas Instruments (TI) K2H SoC platform, 669–670
Texas Instruments 8800 Software Development Board (SDB), 755–765
  block diagram, 756
  components, 756
  control operations, 757
  counters, 759
  external environment, 762–763
  microinstruction format, 757–758
  microsequencer, 757–762
  microsequencer microinstruction bits, 761
  registered ALU, 762–765
  registered ALU instruction field, 764–765
  registers, 759
  stack operations, 759–760
  subfields, 760, 761
Third generation of computers, 18–24
  DEC P DP- 8, 23–24
  IBM system/360, 22–23
  microelectronics, 19–22
32-bit Thumb instructions, 482
Thrashing, 138, 299
Thread, 629, 690
Thread blocks, 690
Threading granularity, 663
Threading strategy
  coarse-grained, 663
  fine-grained, 663
  hybrid, 663
  simultaneous multithreading (SMT), 667
  Valve game threading, 663–665
Thread-level parallelism, 662
Throughput, 71
Thumb instruction set, ARM, 479–481
Thunderbolt, 263, 265
Time-sharing operating systems (OS), 296–297

Timing
I/O modules, 203, 232–233
magnetic disk, 203
memory system effects on instruction, 601
TinyOS, 31
TLP packet assembly, 114–115
Data field, 115
end-to-end CRC field, 115
Header field, 115
Tracks, magnetic disks, 196–197
Transaction layer, 112–114
Transducer, I/O, 231
Transfer of control operations, 433
Transfer rate, 123
Transfer time, magnetic disks, 201–202
Transistors, 17–18
Translation lookaside buffer (TLB), 301–303
Trap flag (TF), 518
True data (flow) dependency, parallelism, 579–581
Truth table, 374, 378, 403
binary addition, 394
for read-only memory (ROM), 393
64-bit, 393
Turing, Alan, 11
Two-level cache memory, 157–164
characteristics of, 158
locality, 158–160
operation of, 160–161
performance parameters, 161–164
relative dynamic frequency of high-level language operations, 159
Twos complement operation of integers, 331–333, 336, 342–347

U
Ultra Enterprise 2, 71
Ultra-large-scale integration (ULSI), 24
UltraSPARC II processor, 71
Unary operator, 417
Unconditional branch instructions, 482
Unconditional jump, 754
Underflow, 353, 358
Unified cache memory, 149
Uniform memory access (UMA), 640
Uniprocessors, 615–617, 619, 621
Uniprogramming, operating systems (OS), 280
Unit of transfer, 122
Universal Serial Bus (USB), 263–264
Upward compatible, 496
Use bit, 146
User/computer interfacing, OS, 276–278
User-visible registers, 491–493
Utilities, OS, 277
Utility program, 277

V
Vacuum tubes, development of, 11–17
Valve game threading, 663–665
Variable-length instruction formats, 473–477
Variable-sized partitions, 295–296
VAX architecture, 300
VAX instruction format design, 474–477, 479, 537
Vector, 243
Vector floating-point (VFP) unit, 603
Vertical loss, 632
Very-large-scale integration (VLSI), 24
Very long instruction word (VLIW), 632
Video display terminals (VDTs), 230
Virtual address fields, 305
Virtual cache memory, 132
Virtual Interrupt Flag (VIF), 519
Virtual memory, 299–301, 494
demand paging, 299
page fault, 299
page replacement, 299
page table, 300–301
thrashing, 299
Virtual Mode (VM) bit, 519
Volatile memory, 32, 124
Von Neumann architecture, 81–82
Von Neumann machines, 13

W
Wafer, silicon, 21
Warps, 696
Watchdog, 680
Wi-Fi, 266
Wilkes control, 735–739, 746
Winchester disk format, 199
Windows, 18
Words, 14
of memory, 85, 101, 167, 174, 495
packed, 441
Write after read (WAR) dependency, 509
Write after write (WAW) dependency, 509
Write back technique, 132, 146, 260, 516, 562, 565
Write hit/miss, 627
Write mechanisms, magnetic disks, 195–196
Write policy, cache memory, 145–147
Write Protect (WP), 521
Write through technique, 145, 260, 622
Write-update protocol, 624

X
X86 and ARM data types, 422–425
Xeon E5-2600/4600, 255–257
XOR operations, 430
XU (translation unit), 10

Z
Zero-address instructions, 418
Zones, defined, 198

CREDITS

Page 4: "There is remarkably . . . not at the time of design" based on Siewiorek, D., Bell, C., and Newell, A. Computer Structures: Principles and Examples. New York: McGraw-Hill, 1982.

pp. 12–13: "2.2 First: Since the device is primarily a computer. . . . It will be seen that it is again best to make all transfers from M (by O) into R, and never directly from C" based on Von Neumann, J. First Draft of a Report on the EDVAC. Moore School, University of Pennsylvania, 1945.

p. 39: Excerpt from: The NIST Definition of Cloud Computing (42 words). Grance, T., and Mell, P. "The NIST Definition of Cloud Computing." NIST SP-800-145. National Institute of Standard and Technology.

p. 57: Figure 2.5: System Clock. Image courtesy of The Computer Language Company Inc., www .computerlanguage.com

p. 269: Figure 7.20: IBM zEC12 I/O Frames–Front View IBM, Reprinted by Permission. IBM zEnterprise EC12 Technical Guide, SG24-8049. http://www.redbooks.ibm.com/abstracts/sg248049 .html?

p. 540: Table 15.2: Weighted Relative Dynamic Frequency of HLL Operations based on Patterson, D., and Dequin, C. "A VLSI RISC." Computer, September 1982.

p. 634: Figure 17.8: Cluster Configurations based on Buyya, R. High Performance Cluster Computing: Architectures and Systems. Upper Saddle River, NJ: Prentice Hall, 1999.

p. 638: "Lists the following as desirable cluster middleware services and functions . . ." based on Hwang, K., et al. "Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space." IEEE Concurrency, January–March 1999.

p. 652: Table 17.3: Typical Cache Hit Rate on S/390 SMP Configuration. MAK97.

p. 670: Figure 18.8: Texas Instruments 66AK2H12 Heterogenous Multicore Chip. Courtesy of Texas Instruments.

p. 693: Figure 19.3: Floating-Point Operations per Second for CPU and GPU. Image courtesy of NVIDIA Corporation.

p. 695: Figure 19.5: Single SM Architecture. Image courtesy of NVIDIA Corporation.

p. 703: Figure 19.11: Intel Gen8 Slice adapted from Intel Corp. The Computer Architecture of Intel Processor Graphics Gen8. Intel White Paper, September 2014.

This page intentionally left blank

digital resources for students

Your new textbook provides 12-month access to digital resources that may include VideoNotes (step-by-step video tutorials on programming concepts), source code, web chapters, quizzes, and more. Refer to the preface in the textbook for a detailed list of resources.

Follow the instructions below to register for the Companion Website for Stallings' Computer Organization and Architecture, Tenth Edition.

1. Go to www.pearsonhighered.com/cs-resources
2. Enter the title of your textbook or browse by author name.
3. Click Companion Website.
4. Click Register and follow the on-screen instructions to create a login name and password.

Use a coin to scratch off the coating and reveal your access code.
Do not use a sharp knife or other sharp object as it may damage the code.

Use the login name and password you created during registration to start using the digital resources that accompany your textbook.

IMPORTANT:

This access code can only be used once. This subscription is valid for 12 months upon activation and is not transferable. If the access code has already been revealed it may no longer be valid. If this is the case you can purchase a subscription on the login page for the Companion Website.

For technical support go to http://247pearsoned.custhelp.com

This page intentionally left blank

ACRONYMS

ACM	Association for Computing Machinery
ALU	Arithmetic Logic Unit
ANSI	American National Standards Institute
ASCII	American Standards Code for Information Interchange
BCD	Binary Coded Decimal
CD	Compact Disk
CD-ROM	Compact Disk Read-Only Memory
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
DRAM	Dynamic Random-Access Memory
DMA	Direct Memory Access
DVD	Digital Versatile Disk
EEPROM	Electrically Erasable Programmable Read-Only Memory
EPIC	Explicitly Parallel Instruction Computing
EPROM	Erasable Programmable Read-Only Memory
HLL	High-Level Language
I/O	Input/Output
IAR	Instruction Address Register
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
ILP	Instruction-Level Parallelism
IR	Instruction Register
LRU	Least Recently Used
LSI	Large-scale Integration
MAR	Memory Address Register
MBR	Memory Buffer Register
MESI	Modify-Exclusive-Shared-Invalid
MIC	Many Integrated Core
MMU	Memory Management Unit
MSI	Medium-Scale Integration
NUMA	Nonuniform Memory Access
OS	Operating System
PC	Program Counter
PCB	Process Control Block
PCI	Peripheral Component Interconnect
PROM	Programmable Read-Only Memory
PSW	Processor Status Word
RAID	Redundant Array of Independent Disks
RALU	Register/Arithmetic-Logic Unit
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
SCSI	Small Computer System Interface
SMP	Symmetric Multiprocessors
SRAM	Static Random-Access Memory
SSI	Small-scale Integration
ULSI	Ultra Large-Scale Integration
VLIW	Very Long Instruction Word
VLSI	Very Large-Scale Integration

THE WILLIAM STALLINGS BOOKS ON COMPUTER

DATA AND COMPUTER COMMUNICATIONS, TENTH EDITION

A comprehensive survey that has become the standard in the field, covering (1) data communications, including transmission, media, signal encoding, link control, and multiplexing; (2) communication networks, including circuit- and packet-switched, frame relay, ATM, and LANs; (3) the TCP/IP protocol suite, including IPv6, TCP, MIME, and HTTP, as well as a detailed treatment of network security. **Received the 2007 Text and Academic Authors Association (TAA) award for the best Computer Science and Engineering Textbook of the year.**

WIRELESS COMMUNICATION NETWORKS AND SYSTEMS (with Cory Beard)

A comprehensive, state-of-the art survey. Covers fundamental wireless communications topics, including antennas and propagation, signal encoding techniques, spread spectrum, and error correction techniques. Examines satellite, cellular, wireless local loop networks and wireless LANs, including Bluetooth and 802.11. Covers wireless mobile networks and applications.

COMPUTER SECURITY, THIRD EDITION (with Lawrie Brown)

A comprehensive treatment of computer security technology, including algorithms, protocols, and applications. Covers cryptography, authentication, access control, database security, cloud security, intrusion detection and prevention, malicious software, denial of service, firewalls, software security, physical security, human factors, auditing, legal and ethical aspects, and trusted systems. **Received the 2008 TAA award for the best Computer Science and Engineering Textbook of the year.**

OPERATING SYSTEMS, EIGHTH EDITION

A state-of-the art survey of operating system principles. Covers fundamental technology as well as contemporary design issues, such as threads, SMPs, multicore, real-time systems, multiprocessor scheduling, embedded OSs, distributed systems, clusters, security, and object-oriented design. **Third, fourth and sixth editions received the TAA award for the best Computer Science and Engineering Textbook of the year.**

CRYPTOGRAPHY AND NETWORK SECURITY, SIXTH EDITION

A tutorial and survey on network security technology. Each of the basic building blocks of network security, including conventional and public-key cryptography, authentication, and digital signatures, are covered. Provides a thorough mathematical background for

such algorithms as AES and RSA. The book covers important network security tools and applications, including S/MIME, IP Security, Kerberos, SSL/TLS, network access control, and Wi-Fi security. In addition, methods for countering hackers and viruses are explored. **Second edition received the TAA award for the best Computer Science and Engineering Textbook of 1999.**

NETWORK SECURITY ESSENTIALS, FIFTH EDITION

A tutorial and survey on network security technology. The book covers important network security tools and applications, including S/MIME, IP Security, Kerberos, SSL/TLS, network access control, and Wi-Fi security. In addition, methods for countering hackers and viruses are explored.

BUSINESS DATA COMMUNICATIONS, SEVENTH EDITION (with Tom Case)

A comprehensive presentation of data communications and telecommunications from a business perspective. Covers voice, data, image, and video communications and applications technology and includes a number of case studies. Topics covered include data communications, TCP/IP, cloud computing, Internet protocols and applications, LANs and WANs, network security, and network management.

MODERN NETWORKING WITH SDN AND QOE FRAMEWORK

A comprehensive and unified survey of modern networking technology and applications. Covers the basic infrastructure technologies of software defined networks, OpenFlow, and Network Function Virtualization (NVF), the essential tools for providing Quality of Service (QoS) and Quality of Experience, and applications such as cloud computing and big data.

COMPUTER NETWORKS WITH INTERNET PROTOCOLS AND TECHNOLOGY

An up-to-date survey of developments in the area of Internet-based protocols and algorithms. Using a top-down approach, this book covers applications, transport layer, Internet QoS, Internet routing, data link layer and computer networks, security, and network management.